NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

AN AUDIO ARCHITECTURE INTEGRATING SOUND AND LIVE VOICE FOR VIRTUAL ENVIRONMENTS

by

Eric M. Krebs

September 2002

Thesis Advisor: Russell D. Shilling
Co-Advisor: Rudolph P. Darken

This thesis done in cooperation with the MOVES Institute

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2002	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: An Audio Architecture Integrating Sound and Live Voice for Virtual Environments			**5. FUNDING NUMBERS**
6. AUTHOR(S) Eric M. Krebs			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(maximum 200 words)***

The purpose behind this thesis was to design and implement audio system architecture, both in hardware and in software, for use in virtual environments. The hardware and software design requirements were to provide the ability to add sounds, environmental effects such as reverberation and occlusion, and live streaming voice to any virtual environment employing this architecture.

Several free or open-source sound APIs were evaluated, and DirectSound3D™ was selected as the core component of the audio architecture. Creative Labs Environmental Audio Extensions (EAX) was integrated into the architecture to provide environmental effects such as reverberation, occlusion, obstruction, and exclusion.

Voice over IP (VoIP) technology was evaluated to provide live, streaming voice to any virtual environment. DirectVoice was selected as the voice component of the architecture due to its integration with DirectSound3D™. However, extremely high latency considerations with DirectVoice, and any other VoIP application or software, required further research into alternative live voice architectures for inclusion in virtual environments. Ausim3D's GoldServe Audio Localizing Audio Server System was evaluated and integrated into the hardware component of the audio architecture to provide an extremely low-latency, live, streaming voice capability.

14. SUBJECT TERMS Virtual Environments, Audio, Voice Over IP (VoIP), Sound, Spatialized Sound			**15. NUMBER OF PAGES** 197
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

This thesis done in cooperation with the MOVES Institute

Approved for public release; distribution is unlimited

AN AUDIO ARCHITECTURE INTEGRATING SOUND AND LIVE VOICE FOR VIRTUAL ENVIRONMENTS

Eric M. Krebs
Commander, United States Naval Reserve
B.S., United States Naval Academy, 1985

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN MODELING, VIRTUAL ENVIRONMENTS AND SIMULATION

from the

NAVAL POSTGRADUATE SCHOOL
September 2002

Author: Eric M. Krebs

Approved by: Russell D. Shilling
Thesis Advisor

Rudolph P. Darken
Co-Advisor

Rudolph P. Darken
Chair, MOVES Academic Committee

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The purpose behind this thesis was to design and implement audio system architecture, both in hardware and in software, for use in virtual environments. The hardware and software design requirements were aimed at implementing acoustical models, such as reverberation and occlusion, and live audio streaming to any simulation employing this architecture.

Several free or open-source sound APIs were evaluated, and DirectSound3D™ was selected as the core component of the audio architecture. Creative Technology Ltd. Environmental Audio Extensions (EAX™ 3.0) were integrated into the architecture to provide environmental effects such as reverberation, occlusion, obstruction, and exclusion.

Voice over IP (VoIP) technology was evaluated to provide live, streaming voice to any virtual environment. DirectVoice was selected as the voice component of the VoIP architecture due to its integration with DirectSound3D™. However, extremely high latency considerations with DirectVoice, and any other VoIP application or software, required further research into alternative live voice architectures for inclusion in virtual environments. Ausim3D's GoldServe Audio System was evaluated and integrated into the hardware component of the audio architecture to provide an extremely low-latency, live, streaming voice capability.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

First and foremost, I would like to thank my wife and children for putting up with the hectic schedule, lost weekends, late nights and my generally "occupied" nature during the this work. Without their patience and support, I would not have made it.

LCDR Russ Shilling, as my primary thesis advisor, was absolutely instrumental to my completion of this course of study. His steadfast support and guidance made this thesis not only educational, but also fun. From purchasing equipment on short notice to our Friday afternoon thesis "discussions", he was always there for me. Dr. Rudy Darken, my Co-Advisor, was equally as important to my success. Through four classes and this thesis, Rudy constantly motivated me to think about my "product" from the standpoint of the consumer - I am confident it is better and it is in no small part due to Rudy's support and encouragement. Lastly, Erik Johnson, as Project Lead for the NPS-developed GFLIB virtual environment library, was crucial to the software implementation portion of this thesis. Hired as a code writer, Erik has been one of the best teachers I have ever had. To these three and the many others who supported and helped me through this process, my heartfelt thanks.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

In the military, most new training systems under consideration today involve implementation of a virtual environment. As real world training locations become scarce and training budgets are trimmed, training system developers look more and more towards virtual environments as the answer. Virtual environments provide training system developers with several key benefits - reconfigurability, variability in training scenarios, and training for distributed teams and individuals. Virtual environment training systems may range from large, complex systems like CAVEs (Cave Automatic Virtual Environment) to Head Mounted Display (HMD) systems to single desktop PCs utilizing videogame technology. Regardless of the type of system implemented, a virtual environment training system must meet both the task requirements of the trainer and the educational needs of the trainee.

A. SOUND IN VIRTUAL ENVIRONMENTS

The design of virtual environment training systems should start with task analyses describing the fidelity and types of sensory cues necessary in the virtual environment to provide a positive training transfer to the real-world task. Most training systems do not require modeling every sensory cue. For example, a virtual environment training system designed for part-task navigation training may not require any audio or haptic cues. Visual displays and cues may be all that is necessary to provide a trainee with enough fidelity to make the part-task training viable and effective. Conversely, an aircraft simulator or Close Quarters Battle (CQB) virtual environment may require implementation of all available sensory cues to be effective and provide a positive transfer of training. The presence of multi-sensory cues in these virtual environment training systems may be necessary to provide the user with a sense of presence and immersion in the virtual world.

Many task analyses do not adequately determine which auditory cues are necessary to create the appropriate level of immersion and sense of presence. More often than not, the absence of an auditory cue will be far more noticeable to a participant in a

virtual environment than its presence. A specific type of task analysis, called an Auditory Task Analysis may be required for complex tasks such as CQB to determine the exact nature and types of auditory cues that induce presence. Significant research is ongoing at the Naval Postgraduate School in several areas relating to the relationship between auditory cues and the sense of presence. Studies examining physiological manifestations relating to presence, auditory cue effects on memory and retention, and a CQB auditory task analysis all relate to the larger issue of providing the most immersive virtual environment possible. For a virtual environment training system to be truly effective, a thorough task analysis, including an auditory task analysis, will confirm whether auditory cues are necessary. For those training systems that a task analysis indicates audio cues are necessary, critical design attention must be given to the type and quality of the auditory cues and the design and implementation of the audio architecture created to deliver those cues.

While system developers have long understood the necessary linkage between display graphics fidelity and the level of immersion, auditory fidelity has not been a principle focus in immersion research or system design. Many virtual environments have been designed and constructed with no thought at all to the implementation of an audio delivery system or audio design principles. We contend that failing to address the audio component of a virtual environment may result in a far less effective training device.

B. RESEARCH OBJECTIVE

The principle objective of this research was to design and implement a fully immersive audio architecture that could be incorporated into any training system. To be fully immersive, the audio architecture must provide:

- Spatialized sound. Spatialized sound refers to sounds emanating from point sources surrounding the listener. This is how sounds occur in the real world. If a virtual environment is to mimic the real world for a given task, sounds related to that task must appear as they do in the real world. Spatialized sound may be implemented with a multi-speaker system such as 5.1 surround, or it may be implemented with a headphone-based spatial audio system that provides both elevation and azimuth cues.
- Modeling of acoustic properties in the environment. Acoustics is a term used to describe audio effects that occur in the real world in a given

situation. For example, sounds inside a room reverberate and reflect around the listener. Depending on the material properties of the room, the effects may be more or less pronounced. The audio architecture used in virtual environments must be capable of mimicking those effects.

- Live voice. Many virtual environment training systems are designed to support multiple participants or teams. Verbal communication between participants or team members may be a critical element to task and mission success. The audio architecture must support live voice between participants or team members. Latency considerations must be taken into account in the transmission of the live voice signal. If the latency in transmission becomes too great, voice communications will not only be ineffective, but also interfere with the training being conducted. While live voice may not be included in every training system, the architecture must be capable of supporting verbal communications for those applications where it is necessary.
- The ability to create a highly detailed auditory environment. The entertainment industry has long recognized the importance of properly designing sound effects and sound systems to add realism, emotion, and a sense of immersion to film and to video games. The first rule of sound design is, "see a sound, hear a sound." (Yewdall, 1999) The audio element, which is probably the most important aspect of evoking the sense of immersion in a VE, is ambient sound. If the appropriate background sounds (machinery, artillery, animals, footsteps, etc) are not included in the virtual environment, the participant will likely feel detached from the action.

The motivation for this research was personal. As a military aviator with seventeen years of experience, the author has been exposed to numerous simulators, virtual environments and training systems that incorporated substandard audio designs and implementations. Each of these systems failed, in varying degrees, to provide an optimum training environment due to a lack of attention to the critical audio component of their systems. Each of these training systems could have easily been improved if an appropriate audio architecture had been incorporated.

Prior to the implementation of our audio architecture, criteria were established for reviewing existing architectures and guide design decisions:

- Commercial-of-the-shelf (COTS) technology. To the greatest extent practicable, the audio architecture should utilize COTS products, both hardware and software. Using COTS technology not only reduces cost of design and implementation, but in many circumstances, COTS technology enjoys industry technical and maintenance support.

- Cost. Cost of the architecture should be minimized when possible by using public-domain software, open-source development APIs, free commercial software development kits, and existing audio hardware.
- PC-based. Most virtual environment training systems under development today will be delivered on PCs. The audio architecture must be able to work within the confines of a standard PC or, if necessary, add a limited amount of hardware to the overall footprint of the training system. Most military virtual environment training systems are being designed to be deployable onboard ships. The audio architecture and system size must not negatively impact deployability.

The second objective of this research was to compare Voice over Internet Protocol (VoIP) technologies with high-end audio systems capable of live streaming voice. VoIP utilizes an encoding-decoding compression algorithm over a standard network connection to provide live streaming voice for any application. Since VoIP uses network connections, network latency and compression latency can vary widely and impact the effectiveness of the live voice component of the audio architecture. Hardware implementations for live voice may overcome the network limitations of VoIP, but may add considerable size to the footprint of the audio architecture, limit the dispersion of the participants in the virtual environment, and increase the cost of the overall audio delivery system. Finally, comparisons were made to evaluate the quality and latency in voice transmissions using VoIP versus an alternative hardware implementation for live voice.

C. THESIS ORGANIZATION

This thesis is organized into the following chapters:

- Chapter I: Introduction. This chapter provides an overall outline of this thesis and describes the research objectives and motivation behind this research
- Chapter II: Background. This chapter reviews existing research into spatialized hearing, sound and emotion, sound design in the entertainment industry, and Voice over Internet Protocol (VoIP) technologies.
- Chapter III: Architecture Design. This chapter reviews current audio software and hardware architectures.
- Chapter IV: Implementation. This chapter describes in detail an audio architecture and implementation for virtual environments using a combination of open-source application program interfaces (API) and high-end, commercial-off-the-shelf (COTS) audio systems.

- Chapter V: Voice Latency Analysis. This chapter describes an experiment to determine latency in voice transmissions between VoIP and high-end audio systems.
- Chapter VI: Conclusions and Recommendations. This chapter provides a recommended audio architecture and future work in improving the software and hardware potions of this architecture
- Appendices:
  - A. GFAUDIO Documentation
  - B. GFAUDIO Source Code
  - C. AUSERVERLIB Documentation
  - D. AUSERVERLIB Source Code

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND

A. SPATIAL HEARING AND SOUND

Spatialized sound in a virtual environment involves "virtualizing" the location of a sound source relative to the listener, manipulating it so that it seems it to be emanating from the desire location. This is accomplished by controlling two factors, the direction from which the sound is coming, and the perceived distance from the sound source (Figure 1).

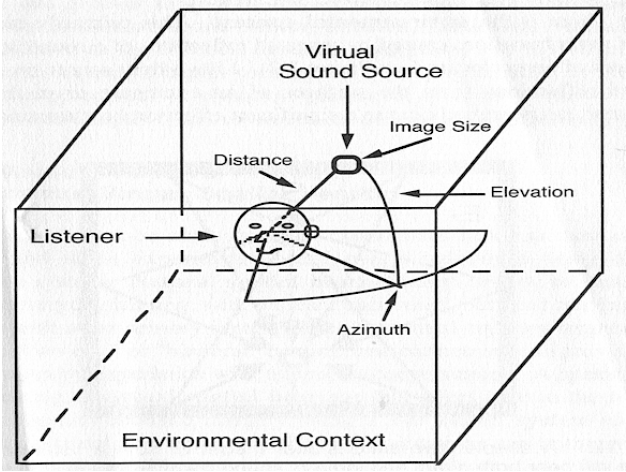


Figure 1. Taxonomy of Spatial Manipulation (From the Operator's Perspective), or of Spatial Hearing (From the Listener's Perspective). (Begault, 1994).

There are two primary methods the brain uses to determine location and direction of sound sources in our environment. For sources at low frequencies (generally below 2Khz) interaural time difference (ITD) is a method by which the brain uses time of arrival differences between the two ears to discriminate direction of the source. Sounds directly in front of the listener arrive at the ears at roughly the same time, while sound waves from sources to the left and right of the listener at different intervals. At higher frequencies, the brain uses interaural intensity (IID) differences to discriminate between sound sources. Many factors play a role in interaural intensity differences: head size, pinnae size and shape, body shape, etc. In general, the brain can discriminate source location by internally recognizing the intensity differences as the sound wave arrives at the ears. Together, interaural time differences and interaural intensity differences provide

the basis for our ability to discriminate source location. Of note, our auditory system does not possess the accuracy of our visual system - the visuals system is capable of discriminating approximately 1 arc minute of difference, while our auditory system is at best only capable of discriminating approximately one degree (Blauert, 1974).

Unlike the visual system, which is always centered on the field of view, the auditory system is continuously functioning in three dimensions. Our accuracy of discrimination varies in both the horizontal and vertical dimensions. On the horizontal plane, we are most accurate at discriminating source locations directly forward or behind where the listener's head is facing. Accuracy degrades as the source location is moved away from the central axis of the head. In the vertical plane around the head, sound sources are symmetric with respect to interaural time difference and interaural intensity differences, and are thus much more difficult to discriminate location. Blauert combines both of these concepts into a phenomenon called "localization blur" (Blauert, 1974). In using interaural time differences and interaural intensity differences at the primary discrimination techniques, our brains can be relatively easily fooled into thinking sounds emanate from incorrect source locations (Figure 2).

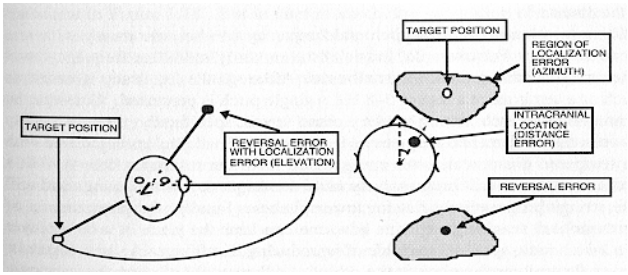


Figure 2. Reversal Error and Localization Error (Blur). (Begault, 1994).

For example, a source on the horizontal plane at 45 degrees relative to central axis of the head would present the same interaural intensity difference and interaural time difference as a source 135 degrees relative to the central axis of the head. When this arises, the primary way we attempt to discriminate these two sources is to use head movements to change the relative positioning of the sound source. Even minor one-degree head directional changes can produce the ITD's and IID's necessary to discriminate direction.

In virtual environments using loudspeakers, the speakers act as the sound sources. The fidelity of spatialization is dependent on the number and location of the speakers in the environment. With as few as two speakers, virtual sound sources can be created over a wide field around the participant, albeit not completely surrounding the listener. Most speaker systems are designed to exist in the horizontal plane around the listener. It is difficult to present sound sources possessing elevation either above or below the listener. Obviously, more speakers create more sound sources, and if the elevation of the speakers is adjusted properly, elevation in virtual sound sources can be achieved. To achieve elevation cues, separate audio channels and speakers would be necessary for each sound source to be presented. For example, sounds virtually positioned above the listener would only play through a speaker system elevated above the physical location of the listener. In an immersive audio environment with many sources, this is not feasible. In general, elevation is not considered in the design of virtual environment audio systems.

With headphones, sound sources can be spatialized in three dimensions using filters incorporating Head Related Transfer Functions (HRTFs). HRTF's are complex filter sets that capture the filtering properties of the pinnae including IIDs and ITDs. Several commercial applications have been developed to permit collection of HRTF data, including the Ausim3D HeadZap HRTF collection system. The HRTF is collected by measuring the IID and ITD of a sound source that is sequentially positioning around the head, using Fourier transforms to build a set of equations that act as a filter to any sound source prior to presentation (Begault and Wenzel, 1993). Once calculated, this filter, which is frequency dependent, can be applied to all sound sources delivered over headphones, providing a fully spatialized, three-dimensional auditory environment.

Our auditory system is far less capable than our visual system in terms of positional discrimination, given a constant audio signal. A virtual environment auditory display does not require exact sound source positioning. The tradeoff between processing efficiency and sound source positioning should be weighed carefully. The auditory display must permit the user to use head movements to discriminate source locations. This is automatic in a virtual environment using loudspeakers. In a virtual environment using headphones, the participant must be head-tracked to permit natural head movements to aid in sound source position discrimination.

For virtual environments using loudspeakers, the concept of the "sweet spot" is a critical design issue. The sweet spot is the place between the speakers where all of the speaker channels combine to create the desired special sound effect. A person positioned in the "sweet spot" is going to benefit from much richer and generally better sounding audio. The "sweet spot" for a two-speaker setup is the mid-point of the line drawn between the two speakers. Additionally, the speakers should be aimed directly at the head. For surround sound systems, the "sweet spot" is the point where the diagonals cross from opposite satellite speakers. The placement of the subwoofer is not critical - the human ear cannot distinguish the direction of the lower frequencies associated with subwoofers.

B. SOUND AND EMOTION

1. Linking Performance with Optimum Stress or Arousal

A large body of research has been conducted into how humans react to stress. Stress, and most importantly an increase in stress, can be caused by external stimuli, such as noise, temperature change, or time pressure for task completion. Stress can also result from internal psychological factors, such as anxiety, fatigue, frustration, and anger. According to Wickens (1999), stress takes on three manifestations:

- Stress produces a phenomenological experience and often an emotional one. The individual subjected to the stress notices a feeling of frustration or arousal.
- Often, a change in physiology is observed, such as a change in heart rate or blood pressure. This affect may by of limited duration or long-term, sustained affect.
- Stressors affect the characteristics of information processing. The assumption is that the affect is degradation in information processing capability for the person subjected to the stress, but this is not always the case.

An easy way to measure the effect of stressors in the environment is through physiological measurements, such as heart rate, blood pressure, pupil dilation, and other activities of the central nervous system. Each of these can describe one's level of arousal. Different types of stressors can either increase or decrease the level of arousal. Stressors such as anxiety, pressure from superiors, noise (a form of audio) generally

increase one's level of arousal while stressors like fatigue will decrease the level of arousal. The Yerkes Dodson Law (Yerkes & Dodson, 1908) characterized the relationship between arousal and performance as seen in Figure 3.

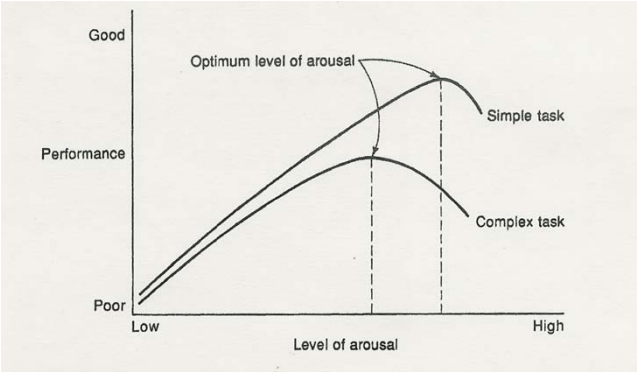


Figure 3. Yerkes Dodson Law. (Wickens, 1999).

As arousal increases towards the optimum level, performance increases. However, once a critical stress threshold is reached, performance will start to degrade. Since stressors can affect the level or arousal, stressors can affect the level of performance. In virtual environments, we have control over many stressors placed on participants, including the fidelity of the information provided the to virtual environment participant. Many stressors will be uncontrollable by the designers of the virtual environment, such as the participant's level of anxiety, pressure by superiors, and other factors in the participant's life. However, for those stressors we can control, we can affect changes in the participant's level of arousal and thus the level of performance. The inclusion of audio in a virtual environment is one stressor that we can control. Proper audio design and implementation in a virtual environment can provide a method to increase the level of arousal and thus the level of performance while in the virtual environment.

2. Linking Arousal with Audio

Research at the Naval Postgraduate School was recently conducted investigating how physiological indicators could be used as a measure of presence in a virtual environment (Scorgie and Sanders, 2002). Subjects were asked to participate in a first-

person shooter video game. The subjects were divided into audio treatment groups as follows:

- Control: No audio presented to subjects
- Treatment 1: Only headphone audio presented to subjects
- Treatment 2: Only loudspeaker audio presented to subjects
- Treatment 3 - headphone and subwoofer speaker audio

The physiological data that was measured included heart rate, blood volume pump (BVP), galvanic skin response, and temperature.

While the intent of the experiment was to determine whether physiological markers or indicators could be used as a measure of the sense of presence, the data can be used to support the link between arousal or stress and audio for this paper. Electro dermal activity, heart rate and blood pressure all increased with the inclusion of audio in the virtual environment, indicating increased arousal. Body temperature decreased with the inclusion of audio; this is to be expected - the body withdraws blood from the extremities and sends more to the vital organs during higher states of arousal, normally associated with a "fight or flight" syndrome. All effects were found to be statistically significant. The results of the experiment indicate that the addition of audio, in whatever form (treatment group), raised certain physiological measures, all of which can be directly related to one's level of arousal.

C. SOUND AND TRAINING

1. Linking Audio with Performance

Performance is linked to an optimum level of stress or arousal one is subjected to in his environment. It has been shown that the inclusion of audio in an environment can raise or affect the stress or arousal level of a person in that environment, whether it is the real world or a virtual environment. A transitive link therefore can be established between performance and audio. The presence of audio cues can raise the arousal levels of participants. If the arousal level of the individual can be elevated to the optimum level, performance will improve. It stands to reason, therefore, that including audio cues appropriately should improve performance. To this point, performance has not been

linked to any task or environment. The next three sections will further examine the affect on performance in the context of a training environment. This is not exclusive to virtual environments - it applies to real-world training environments as well. However, if the link is established between audio and training performance in any environment, at stands to reason that it will apply also in virtual environments. This is the key to linking audio cues that may not be evident as necessary in a task analysis into a requirements specification of a training system.

2. Linking Performance with Memory, Expertise and Training

Wickens and Holland (1999) divide memory into two main categories - working memory and long-term memory. Working memory is the short-term storage capacity where we temporarily store, analyze, transform and compare data. Long-term memory is where information is stored for later retrieval and use. It is practically, if not scientifically, considered to be of infinite storage capacity. Wickens suggests there are five characteristics of how we learn, only two of which will be covered here. First, there is am emphasis on instances or situations. We generally learn through specific examples or situations and then generalize to others. This theory is supported by Brown, Collins and Duguid (1989) in their paper on Situated Cognition. Their research indicates that humans learn best is the context of where the learned material or subject matter can be used. They coin the phrase "cognitive apprenticeship"; learning through doing in a specific situation is far more effective than learning in an abstract, classroom environment without tangible examples. The second characteristic of learning is that we use chunking, or grouping of associated individual pieces of data into larger single units for easier recall and access.

Posner (1983) and Chi, Glaser, and Farr (1998) suggest that the defining characteristic of expertise is exceptional feats of memory, both in chunking strategies and recall. Training systems, including virtual environments, are created to improve performance in a given task, hopefully to the level that of an expert. If expertise is gained through repetitive practice, elaborative rehearsal and exposure to multiple experiences in a domain, a properly constructed training system will afford trainees the

requisite experiences to gain that expertise. Does the inclusion of audio assist the trainee in the acquisition of domain knowledge, skills, and improved memory?

Dinh, Walker, Song, Kobayashi, and Hodges (1999) conducted an experiment to determine if the presence of multi-sensory inputs in a virtual environment had any impact on the user's level of presence and memory retention for objects and events in that environment. They found that the addition of sensory inputs, other than the standard visual modality provided with most virtual environments, significantly increased the user's level of presence in the virtual environment. In basic terms, an increase in the level of presence was found for each type of sensory input added - tactile, audio, and olfactory. Each had an additive effect on the level of presence. Their study supports the contention that the addition of audio increases the sense of presence. Furthermore, the results of their study indicate that memory of objects and events in the virtual environment improved as additional sensory inputs were provided to the user. A virtual environment that uses a single modality, such as a visual display, is inferior to one that incorporates multi-sensory information, including auditory, tactile and olfactory cues. Admittedly, not every task requires multi-sensory cues, and not every virtual environment training system will be capable of multi-sensory cueing information. However, their study further justifies conducting thorough task analyses to determine whether the presence of multi-sensory cues is appropriate. When task analyses indicate that audio cues are necessary for a particular task, inclusion of these audio cues in the virtual environment should, theoretically, improve memory, develop increased expertise in the specific domain, and improve task performance.

The two previous paragraphs transitively link audio in an environment to an increase and improvement in performance. First, audio was linked with an increase in arousal. Then, an increase in arousal was linked with an increase in performance. This link suggests that proper audio design and implementation can improver performance in virtual environments.

3. Selected Task Analyses

This section will examine several task analyses developed to provide the foundation for virtual training environments. Each is assessed for the following:

- Tasks analysis requirement(s) for audio cues
- Whether those audio cues are could be included in the virtual training environment
- The possible benefit of including audio and audio cues in the virtual environment

Grassi (2000) conducted a task analysis of pier side ship handling as a development tool to construct a virtual environment pier side ship-handling simulator. In the critical cue inventory for determining whether a ship's engines have properly started and are online, a critical cue for the conning officer is the sound the conning officer will hear as the ship's engines start up (Figure 4). During a conning officer's assessment of environmental conditions and surrounding environment, auditory clues provided by ship's pennants flapping in the wind provide the conning officer with an estimation of wind speed and direction.

Critical Cue Inventory for	
Phase: A	**Subgoal:** Verify_Engines_Are_Started_And_Online
CUE	***DESCRIPTION***
Look at status board on bridge	Used to verify engines have been started. The conning officer will visually look at the status board on the bridge to confirm whether the engines are started. This board, constantly updated by an enlisted watch stander who receives reports from all over the ship via sound powered phones, displays the status of all critical components onboard the ship including the engines that are online and the steering units energized.
Hear engines start up	Used to verify engines have been started. The conning officer will often hear the whine of the engines as well as the sound of exiting smoke and steam from the smoke stacks.
See smoke come out of smoke stack	Used to verify engines have been started. The conning officer will visually look for increased flow of smoke out of the smokestack. Although there may be some smoke exiting due to other components being operated, such as the ship's Service Generators, as an engine is brought online, or started, there is a dramatic increase in the amount of smoke billowing from the smokestack.

Figure 4. Example of Critical Cue Inventory for Pier Side Ship Handling - Verify_Engines_Are_Started_And_Online. (Grassi, 2000).

Numerous references are made to the conning officer checking rope lines for taught or slack conditions in many of the individual task elements. The sound of a line stretching to near breaking point is a well-known auditory cue of a line being too taught. As orders are given by the conning officer to maneuver the ship, especially those involving changes in engine state or RPM, answering bells provide the officer with an

auditory cue as to order compliance. Each of these is listed as a critical cue. If a virtual environment simulator for ship handling ignores these critical cues, potentially degraded training in pier side ship handling will result. Conning officers trained in pier side ship-handling in the virtual environment will not gain the experience of using those critical auditory cues in actual ship-handling in the real world.

Similarly, Norris (2000) refers to specific audio cues in his task analysis of underway replenishment ship handling. Like Grassi, he finds engine sounds as critical audio cues in determining whether verbal orders for maneuvering and speed changes have been carried out. During underway replenishment, the bridge of a ship is a host to many activities, many of which are driven by verbal commands. Verbal communication is critical to every phase of underway replenishment (Figure 5). Verbal interaction between the Commanding Officer, conning officer, Helm, and Navigator is constant and precise - each monitors the communication between all of the other players in an underway replenishment situation.

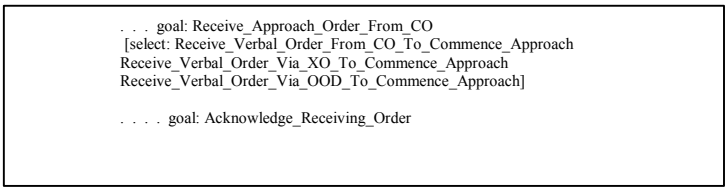


Figure 5. Example of Underway Replenishment Verbal Audio Cues. (Norris, 2000).

Norris' task analysis is replete with critical verbal cues between all participating watch stations. A virtual environment simulator modeling underway replenishment that did not include verbal communication would be severely limited.

Both example task analyses indicate that specific audio cues are necessary to complete the task successfully. Both scenarios could be incorporated into virtual environments without any audio at all. Text messaging could be a possible substitute for voice audio and ship's engine noises could simply not be included. However, the task analysis suggests that these cues are critical to the success of the evolution. Ignoring critical cues degrades the quality of the simulator and training. Couple the "critical-ness" of the cues with findings that multi-sensory cues in virtual environments add in memory

retention of objects and events, inclusion of audio in the virtual environment simulator is seen as not only beneficial, but necessary.

4. Summary

A transitive link has been established between the inclusion of audio in virtual environments with a benefit to both training and performance. This link has not been established to be causative or direct, but instead built upon existing research into memory, arousal, expertise, training and learning. Further study is warranted, and transfer-of-training experiments should be conducted to determine if there is an actual cause-effect relationship between virtual environment audio and training performance. However, given the relationships described above, virtual environment developers and designers should pay significant attention to the audio design and implementation of their systems. Ignoring the link between audio implementation and training performance can result in degraded training performance and the ultimate validity of the training system. As a result, degraded training will affect operational performance. In the military, where the skills being trained may be of a life or death nature, operational performance is of utmost importance.

D. VOICE OVER IP (VOIP) TECHNOLOGY

VoIP is the transportation of speech signals in an acceptable method from sender to destination over an Internet network. The speech signal is digitized pieces of voice conversation sampled at regular intervals. These samples are sent via the network to the desired destination where they are reconstructed into an analog signal representing the original voice. In a networked virtual environment, VoIP offers a technology to permit live voice between large numbers of networked participants. In the military, many virtual environments simulate training scenarios involving multiple team members working together to complete a mission or a task. If the “real world” task or mission is accomplished through voice communications, using text-based messaging for live voice communications will be cumbersome and unnatural and will degrade the realism of the training system.

Using an Internet protocol (IP) network requires the utilization of an IP protocol for transmitting the information. IP networks are sometimes referred to as "packet" networks, for they communicate through the sending and receiving of data packets with known formats. Two standard protocols, TCP/IP and UDP are the most widely used protocols used today. All Internet Service Providers (ISP) support TCP/IP. Everyone with a home dial-up Internet account, home Digital Subscriber Line (DSL) account or home cable modem Internet account uses TCP/IP for communications with the Internet. TCP/IP refers to the format of data that is transmitted over the network and the rules in force for ensuring delivery at the desired location. TCP/IP is considered to be "reliable" - reliable means that each individual packet that is sent over the network is verified at the receiver and acknowledged. If the data is larger than a single packet, it would be broken down into several individual packets and each transmitted separately. Packets are reassembled in the proper order at the destination prior to delivery to the client's application. TCP/IP guarantees that packets will be reconstructed at the receiver in proper order. Reconstruction in the proper order is of vital importance to a voice signal. Out of order or lost packets will significantly degrade the quality of the transmitted voice. However, the processing overhead and delay for this guarantee will significantly increase latency in transmission and reconstruction of the voice signals.

UDP is the second-most widely used IP protocol in use. Unlike TCP/IP, UDP is unreliable. The UDP protocol does not contain the stringent requirement to acknowledge each individual packet. Packets are transmitted from the sender and essentially forgotten. While this reduces the overhead and delay in processing, packets can arrive out of order or be dropped from reception completely. Both of these protocols use an IP network for transmission. IP networks do not guarantee a specific path for delivery of packets between sender and receiver. Each packet may take a different network path. For this reason, UDP is generally considered unsatisfactory for live voice.

The latest IP protocol developed specifically for streaming audio and video over the Internet is Real-Time Transfer Protocol (RTP). RTP imposes packet sequencing and time stamping on a UDP data stream to ensure sequential packet reconstruction at the receiver while not imposing the high processing overhead of reliable transmission.

The core components of a VoIP system for virtual environments will be slightly different from that of a VoIP system designed for other purposes, such as an alternative to a telephone. The primary difference will be the inclusion of spatialization processing for the transmitted voice signal. This section will not address spatialization of VoIP; refer to Chapters III and IV for specifics on how spatialization is incorporated into VoIP. The entire process of the core VoIP system is depicted in Figure 6. The arrows that point downward define the path that is followed when sending speech signals; the arrows that point upward define the processing sequence when speech signals are received. When the label of a box contains two items, the left one is refers to the sending of speech signals and the right refers to the reception of such signals. They are grouped together because they operate at the same level: the right item does approximately the opposite of the left one.

In order to send a live voice across a computer network, the speech signal has to be digitized prior to transmission. In most VoIP applications, the PC sound card performs the sampling of the speech signal and conversion to digital information.

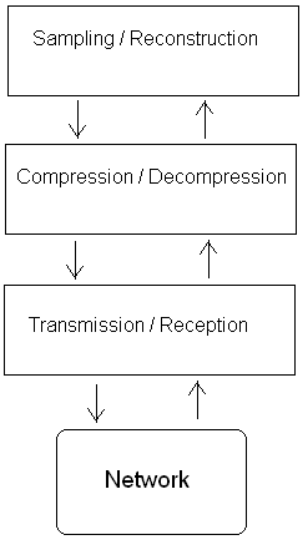


Figure 6. VoIP Components.

When a digitized block is received, it has to be transformed back into an audio signal. Like the sampling and digitization for transmission, the transformation is accomplished by the receiver's PC sound card. The output of the process is played either

through speakers or headphones for the listener. In essence, reconstruction is the reverse operation of sampling.

Several issues have to be considered before transforming the digitized signal. First, if multiple persons are allowed to talk at the same time, as would be the case in many virtual environment training systems, the speech signals of those persons have to mix together at the receiver. The mixing is also done on the receiver's PC sound card.

Second, when sending blocks of data across a network, there will be tiny variations in the time it takes each block to get to the destination; unfortunately, these variations can become quite large. The path each packet takes over the network from sender to receiver is not controlled by either (assuming an Internet connection between sender and receiver). There is a large body of research into packet flow over distributed networks, which will not be a focus of this investigation. It suffices to say that once a packet leaves the sender, control over its path to the receiver and the time it takes to arrive at the receiver, is uncontrollable. The only way to control network flow is to control the entire network. In the case of the Internet, this is impossible. In military training systems, this may actually be possible, if the networked simulation is over an IP-based network detached from the Internet. The problem with variation can be seen in the following example. Suppose the voice signal is reconstructed in the exact sequence the voice packets are received. Because of the variation in arrival time due to uncontrollable network pathing, it is possible that either the next block has not yet arrived when the output of the first one is finished or the blocks arrive out of sequence. To overcome this, the reconstruction scheme purposely creates a delay to allow received packets to accumulate. The accumulated packets can be re-sequenced not in the order of arrival, but in the order of transmission. However, buffering introduces a significant delay in the presentation of the voice signal.

Further complicating the matter, the digitized information requires a certain amount of the available bandwidth of the connection. Sampling and reconstructing the speech signal at an extremely high rate is possible, especially with the advances in computing technology of recent years. However, higher sampling rates have a tendency to flood the network connection with data packets. Every network suffers from some

limitation in bandwidth. Simply increasing the sampling rate will not necessarily reduce latency and can other detrimental effects on network performance. In a networked virtual environment the voice signal is not likely to be the only data transmitted over the network. Environment state data, participant actions and controlling information must continually be sent to maintain a shared state between the separate virtual environment users. All network traffic, including the voice data, must share the bandwidth of the connection.

Very often compression schemes are used to reduce the required bandwidth for voice communication. Compression refers to encoding the information with an algorithm that reduces the raw data into smaller pieces. Several types of compression strategies exist. Some use compression techniques, which are also used on other kinds of data. Others are designed specifically for streaming media over the Internet. These types of compression significantly reduce the amount of data that must be transmitted. However, the more intricate and processor-intensive the algorithm used for compression, the greater the delay in transmission of the original voice signal.

Once the compressed blocks with speech data reach the destination, they have to be decompressed. The decompression is very closely related to compression. It is the inverse operation of the compression scheme that was used. Compression and decompression are very important when the network connection is slow, as with dial-up connections to the Internet.

Up to this point, transmission of voice packets has been presented in the form of a single sender and single receiver. In virtual environments, many training scenarios will involve many more than two participants. A growing bandwidth problem is created if the sender must individually transmit voice data to all other participants, especially if the number of participants is large. As the number of participants grows, an exponential number of paths between participants present itself for transmitting the voice information blocks. Statistically, this is referred to as an $O(n^2)$ problem. Multicasting is IP-based protocol where individual machines (users) subscribe to an IP address as if that address where another user. Individual users transmit and receiver from the multicast address. The difference in this scheme is that each user receives all data transmitted to the

multicast address. This permits a single transmission of the voice signal and simultaneous reception by all users. Multicasting promises to improve streaming media over the Internet by reducing the number of messages sent to large numbers of subscribers. However, multicasting requires modifications to hardware at all of the switches and routers within then network path. Changes that would permit multicasting over the Internet have not taken place. Small-scale networks, where control and access to the network hardware is possible, offer the only currently available option for multicast use.

E. LIVE VOICE IN VIRTUAL ENVIRONMENTS

In "Adding Voice to Distributed Games on the Internet", Bolot and Fosse-Parisis (1998) explore several of the issues in adding a spatialized live voice capability to a distributed virtual environment. There is evidence to suggest that this capability and audio fidelity may be needed for positive training in certain complex applications, such as Close Quarters Combat (Greenwald, 2002). Greenwald found that verbal communications in three of four phases of CQB were evaluated as the highest priority auditory cue necessary.

Echo is a major concern when two or more players join in a live voice session. Echo occurs when each participant's microphone collects not only their voice transmission, but also the playback of the other participants. They analyzed the amount of CPU utilization to implement echo suppression or canceling algorithms, and quickly deduced that their system (a Pentium II 200 MHz) could easily be overwhelmed simply running the computations to reduce echo. Their conclusion that the most effective, and CPU inexpensive, method to reduce echo is to play back voice audio through headphones. Although this study was conducted in 1998, and significant improvements have been made in CPU processing power, this is still a valid recommendation. If live voice is played over speakers, echo will always be an issue that must be dealt with. The authors discuss two additional factors that must be taken into account if an accurate spatialized live voice session is to be modeled. The first is the Doppler Effect, which depends on the relative speed of the participants. The Doppler Effect refers to how sound waves are affected when the source is in motion. As a source moves, its velocity affects

the speed at which sound waves arrive at a listener. If the source is moving towards the listener, the sound waves are compressed relative to source velocity. The perceived sound at the listener is that the frequency has shifted up from the original emitted frequency. Conversely, as the source moves away from the listener, the sound waves are rarefracted relative to source velocity. In this instance, the perceived frequency emitted from the source appears to be shifted down in frequency. Although the original source frequency is never altered, its velocity affects the listener's perception. The amount of frequency change is dependent on source velocity. A simple example of the Doppler Effect is to listen to a train pass by. Initially, as the train approaches, the sound of its whistle will appear higher in pitch, but fall lower as the train actually passes by the listener's position. The second is "transmitter directionality", referring to whether or not the speaker is facing you or turned away from you. Doppler effect need not be calculated by the voice "sender", if some underlying network capability allows for transmission of other data as well. If this capability exists, transmission of data packets encoding the player's velocity can be decoded and Doppler processing can be achieved at the receiver. However, Doppler processing of live voice is generally not required. The velocity required to perceive the pitch change associated with Doppler is much larger than a person can move. Directionality of the voice, modeling the radiation pattern of the voice of the speaker, requires significantly more sophistication. At a bare minimum, speaker orientation and position must be included in network data from sender to receiver. Although processing at the receiver increases to add the directional capability, the inclusion of directivity as an attribute of live voice need not significantly increase network traffic.

The authors raise the critical issue of synchronization and latency - ensuring that the audio and visual presentations are synchronized to a level where the "tolerable de-synchronization" level is not exceeded. Although the tolerable level of de-synchronization is individual-dependent, their findings indicate that there is a generally accepted interval of 185 ms or less, where the desynchronization and latency will be un-noticeable to the user. The issue here is not overall latency in the networked virtual environment. Any networked virtual environment will have some level of latency due to network loading, network transmission path, and other IP-based network issues. The true

issue for live voice is ensuring that the latency in the live voice is no greater than the average network latency inherently present in the system.

Architectural acoustics play a significant role in how live voice is presented in virtual environments. The sound of the live voice should be different depending on the virtual location of the speaker. For example, a voice associated with a participant standing in the middle of a field should sound very different than the voice of a participant standing in a tiled room. Reverberation and reflection quality of the environment should affect the live voice in the same way if affects ambient sounds. However, including or encoding that type of information into the voice data is extremely difficult. As previously discussed, digitizing the voice signal is a time and CPU expensive operation that induces latency. Adding more data, in this case not only the voice but the acoustic effect as well, further burdens the digitization, compression, decompression and reconstruction scheme. A superior implementation will incorporate architectural acoustics in the same manner as voice directivity. Utilizing the network connection established for the voice signal, simple attribute data packets can be sent from speaker to listener to set required parameters on the listener's machine to achieve the requisite acoustical effect.

Does spatializing the live voice really have any benefit? Nelson, Bolia, Ericson and McKinley (1999) conducted an experiment to determine whether spatializing speech in a multi-talker environment aided in detection of a critical phrase. This was an experiment to address the "cocktail party" effect (Yost, Dye and Sheft, 1996). The "cocktail party" effect describes our ability to discriminate between multiple, on-going conversations around us. This is especially applicable in several military applications, such as the Close Quarters Combat (CQB) scenario, currently under development at the Naval Postgraduate School. The author's intent was to examine the number of competing signals that could effectively be discriminated, and also examine any differences between free field and virtual environments. In their experiment, subjects were exposed to from one to eight simultaneous speakers, either spatialized or non-spatialized (Figure 7), and speakers were of both genders.

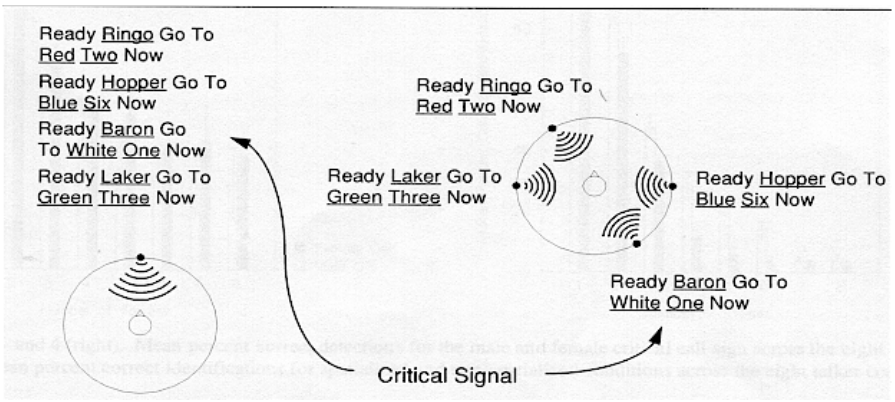


Figure 7. Comparison of Non-Spatially Separated (Left) and Spatially Separated (Right) Audio Conditions. (Nelson, Bolia, Ericson, and McKinley, 1999).

The researchers collected data on detection of a critical signal and identification of additional information that would emanate from the same source position as the target signal. Spatialization had a significant effect on the identification of the signal. The results suggest that our ability to search an auditory environment for cues is significantly improved when those auditory signals are spatialized. However, their study found no advantage when spatializing the speaker's voices when there were more than six talkers. For their study, spatializing voice is most effectively when there are between two and six talkers. Other studies have suggested that the limit on the number of spatialized signals that can be identified is much higher. Regardless, spatialization significantly improves signal detection over non-spatialized signals.

Campbell (2002) conducted an experiment to determine whether spatializing messages in a headphone display would improve recognition and response. He found that a listener could accurately respond to spatialized, overlapping messages 47% of the time. When the overlapping messages where not spatialized, listeners could only respond accurately 17% of the time. By spatializing multiple messages, his results indicate a 30% improvement in the listener's ability to discern overlapping messages. Thus, spatializing live voice in a virtual environment can be beneficial. A virtual training system designed to immerse a participant in a military situation where detection of live voice information is critical is definitely a candidate for spatialized live voice.

Nelson's investigation of spatialized speech signals was primarily based on azimuth separation. Brungart and Simpson (2001) investigated the ability to separate

speech signals when spatialized by distance from the listener in the both the near field (defined as within one meter of the listener) and far field (greater than one meter). In the far field, interaural intensity difference is negligible, since the distance between ears is relatively small compared to the distance between the ears and the sound source. Like Nelson, Brungart and Simpson tested subjects on their ability to discriminate signals and indicate the reception of a target speech signal. The target signal was masked with either random noise or other speech signals. With simple noise, the brain essentially accomplished the discrimination between the two primarily by using Signal to Noise Ration (SNR). The authors call this as energetic masking. When the masking signal is speech, the brain made better use of the binaural cues in discrimination. The authors refer to this as informational masking. Depth of both the target and masking signal was varied, but limited to one meter. Again, this provides a strong argument that if live voice is to be added to a virtual environment training system, spatializing the live voice enhances the user's ability to discern voice information coming from different positions.

These two experiments indicate that spatially separating voices, either by angular separate or distance separation, can improve voice audio discrimination among multiple sources. As previously mentioned, a primary scenario where spatialized VoIP could be employed would be a virtual CQB exercise. In typical room clearing CQB environments, small groups of individuals move in teams through a building. Teams may have contact with each other, and within a team members remain in close proximity. Accurately representing the speech signals through acoustic spatialization in a virtual environment would support training for CQB by creating the same voice audio effects as those found in the real world.

Perhaps the most important indicator of whether spatialized voice has any benefit in a virtual environment is whether or not the participants in that environment feel it increases, decreases or has no effect on their awareness of that environment. Baldis (2001) conducted a study of the effects of spatialized voice on participants in a teleconference. In her study, three configurations for used: non-spatial voice audio, co-located spatialized voice audio (each conference participant's voice audio was played through a single speaker mounted on top of the monitor and all speakers were placed near each other and adjacent to visual images of the conference participants) and scaled-

spatialized voice audio (individual speakers for each of the conference participants placed in a semi-circle around the subject at -60, -20, 20, and 60 degrees relative). Data was collected through the use of tests (memory and comprehension) and post-conference questionnaires. For both the co-located and scaled-spatialized conditions, analyses of the findings indicated that subjects were better able to focus on who was speaking and comprehend the content of what was said. Regardless of the visual presentation, the subjects indicated a clear preference for the spatialized voice audio.

F. ARCHITECTURAL ACOUSTICS

Acoustic modeling is important for design and simulation of three-dimensional auditory environments. The primary challenge in acoustic modeling is computation of the myriad paths from a sound's source position to a listener's receiving position. A sound may travel from source to listener via a direct path, a reflected path, or a diffracted path. In the real world, sound seldom travels from source to listener along only one of these paths. A sound arriving at our ears generally follows many paths that constructively and destructively interfere with one another to create the sounds we hear. For example, an omni-directional sound in a room arrives at the listener via theoretically an unlimited number of paths. While there is only one direct path from source to listener, the number of reflected paths can be huge. The sound can reflect numerous times between wall surfaces before it arrives at the listener. Objects in the room can diffract the sound, creating new paths of reflection. To accurately model architectural acoustics of a room is very difficult and processor expensive. Add to that the fact that most rooms are not simple cubes but multi-faceted enclosures and the problem of accurate acoustics become almost impossible.

There are three general methods employed to integrating architectural acoustics in virtual environment. The first involves pre-computation of acoustic modeling prior to runtime. Reflections and diffractions are computed for every source and listener position possible in the virtual environment and the results can be applied at runtime for a dynamic auditory environment. The second involves auditory ray tracing techniques similar to ray tracing in the visual sense. Large numbers of sound rays are traced from source to listener, sound energies are calculated for transmission path length, and the

result is available at runtime. Neither of these methods is truly interactive, since they both require pre-computations of the acoustic model. The quality of the acoustic model is directly related to the amount of processing allotted prior to runtime

A third solution to providing architectural acoustics is numerical modeling. In numerical modeling, the acoustic effect is obtained not by actual mathematical computations in complex algorithms, but instead based on numerical input to software algorithms that process the sound for a desired effect. There is not in relation to the environment. The effect obtained is a result of numerical input to the effect algorithm. Where do the numerical inputs come from? In the gaming and entertainment industry, the numerical inputs are the result of programmer and audio engineer trial and error. Numerical inputs are varied and modified until the desired effect is obtained. In other words, "work the numbers until it sounds right." While it seems like this methodology might be haphazard, it is the technique most gaming companies have incorporated into the sound design. It is also the basis for how this architecture will attack the problem of acoustic modeling.

III. CURRENT ARCHITECTURE DESIGNS

A. INTRODUCTION

This chapter examines three software application programming interfaces (API) and one high-end hardware architecture implementation. The APIs reviewed are all either open-source, public domain (GNU Public License) or part of free Software Development Kits offered by their respective development companies.

B. OPEN AUDIO LIBRARY (OPENAL)

OpenAL is a cross-platform audio API designed to provide a software developer with a simple, easy to use interface to spatialized audio (available at either http://www.openal.org or http://developer.creative.com). The primary force behind the development of OpenAL is Creative Technology, Ltd. Creative Technology, a company based in Singapore, is a global leader in PC entertainment and audio products. Best known for its SoundBlaster™ and Audigy™ line of PC sound cards, Creative Technology has entered into many joint ventures with PC gaming companies, including Epic Games Inc., a partner in the Naval Postgraduate School's America's Army project. Creative uses OpenAL as their primary audio programming interface for Win32, UNIX, and Linux platforms (Creative does not currently support Mac OS X and provides only minimal support for Mac OS 9 and earlier).

OpenAL is a platform-independent "wrapper" API for operating system on which it operates. For example, on a win32 platform, OpenAL accesses the DirectSound3D™ or DirectSound3D™ driver. Platform operating system discovery is automatic in OpenAL.

A few of OpenAL's capabilities include:

- Audio Contexts - a context in OpenAL can best be described as an audio "situation" - an environment consisting of a listener and sounds. OpenAL, like all audio programming API's only support one context per machine, except in those circumstances where a single computer contains multiple sound cards. In those cases, multiple contexts can be implemented for each sound card.

- Spatialized Audio - OpenAL supports one listener per context, and as many sounds as the host machine memory will support. OpenAL will generally manage whether sounds are processed on the sound card (in hardware) or on the host CPU (in software) automatically. The priority is to utilize hardware assets (buffers) first, followed by software. Most sound cards support between 16 and 64 hardware-processed, simultaneously playing sounds. Spatialization of audio is accomplished by constructing buffers of sound data - OpenAL currently supports multiple audio formats, including PCM wave files and MP3 formatted data. The data is loaded into memory through a simple API call, and the API returns a handle to the data for subsequent playing or looping. The developer has control over the how the sound file is played; playing looping, stopping and restarting, rewinding are all inherent capabilities. Spatialization is accomplished through a simple positioning method that is source-specific for each buffer. The single listener can also be position independently of each sound source. One aspect of OpenAL that is different from other audio API's is the separation of the audio source from the audio data. A source is a buffer that can be positioned - it is not tied directly to any single wave file or sound - the source can be positioned and any wave file can be played through that source. While this may complicate programming for a beginning audio programmer, this distinction is very powerful.
- Audio Rolloff and Attenuation - OpenAL provides for audio rolloff (attenuation of audio sources based on distance from the listener) through three different attenuation models - inverse distance, inverse distance clamped, and exponential. Selection of rolloff model is left to the developer. Manual attenuation of sources can be accomplished through volume settings specific to each source.
- Static and Streaming Audio - OpenAL supports both static buffers (buffers whose data is loaded completely and stored in memory) and streaming buffers (buffers that contain only a portion of the audio data at creation, but continually read new chunks of data as specified intervals). Streaming audio permits the application developer to play extremely large wave files without the memory of CPU penalty of storage and retrieval. Control over the "feeding" of audio data is exposed to the developer.
- Pitch Bending or Frequency Shifting - OpenAL supports frequency modification of audio sources at execution time. This is especially beneficial when modeling sounds such as automotive engines - the pitch of the engine sound can be modified to reflect change in velocity.
- Doppler Processing - OpenAL supports audio source Doppler effects. OpenAL does not calculate audio source velocity and automatically modify source frequencies for the Doppler Effect, but manual setting of velocity parameters will permit Doppler effect processing. One shortcoming to be mentioned here is that, in its current release, OpenAL

does not permit setting of a reference velocity for Doppler calculations. OpenAL has "hard-coded" the speed of sound at sea level (in meters/sec) as its reference velocity. This restrains the developer from being able to amplify the Doppler Effect. While from a physically-based modeling perspective a developer may not be inclined to change the physical properties of a sound, developing audio software from an entertainment perspective may require certain audio effects to amplified or diminished. OpenAL does not support this. A typical example is that of a train approaching and departing a listener's position.

Combining all of the above capabilities into a single, platform-independent API makes OpenAL extremely useful. OpenAL is implemented in many PC gaming engines like Unreal, including "America's Army: Operations" developed at the Naval Postgraduate School. One significant limitation of OpenAL as an audio API is that it does not directly support live voice audio. Live voice audio can be accomplished with OpenAL, but requires the use of 3rd -party API's for voice capture and encoding, network transmission of voice data, and decoding. Once decoded, the voice data could be used as an input to an OpenAL streaming buffer and position accordingly to provide a spatialized voice effect. However, the latency induced by this processing limits OpenAL's utility to virtual environments that do not require live streaming voice.

C. DIRECTSOUND3D™

DirectSound™ and DirectSound3D™ are audio programming API's produced by Microsoft. Originally released as individual API's, they are now integrated and released as a core component of Microsoft's DirectX™ programming suite, currently in release version 8.1 (available at http://www.microsoft.com\windows\directx). DirectMusic™, released for the first time in DirectX 8.0, is a new audio programming API that both wraps DirectSound3D™ and introduces several new functionalities.

DirectSound3D™ as an API contains all of the functionality of OpenAL listed above, with the following limitations:

- Limited to Wave Files - DirectSound3D™ only supports wave file PCM data in its current release. While wave files are the audio programming industry's format of choice, this limitation excludes using other types of sound data, and may require programmers to obtain additional software capable of converting audio files to PCM format.

- Integration of Data and Sources - DirectSound3D™ tightly integrates sound data and the buffer through which it will be played. A static buffer can only be loaded with audio data once. It can be played and repositioned, as many times as the developer desires, but it can never accept new data. In OpenAL, different audio data could be played though the same buffer. For streaming buffers, functionality of OpenAL and DirectSound3D™ is equivalent - audio data can be continually fed into a streaming buffer and replaced.

The following lists additional capabilities that DirectSound3D™ has over OpenAL:

- Audio Effects - DirectSound3D™ supports seven different types of audio effects processing directly within its API - echo, gargle, compressor, chorus, distortion, flanger, and a limited reverb. While these effects are relatively simple and not extremely flexible, they do provide the programmer with access to a limited range of audio effects without having to learn and utilize another API.
- Live Voice - perhaps the single greatest advantage to DirectSound3D™ over any other API is its integration of live voice. For those applications requiring live voice, DirectX™ contains a core module, called DirectPlay™. DirectPlay™ supports networking on a client-server or peer-to-peer structure. A sub-component of DirectPlay™ is DirectVoice™. DirectVoice™ integrates DirectPlay™'s networking with DirectSound3D™'s spatialized audio capability to provide a spatialized live voice capability to any programmer. Currently, DirectVoice™ will support up to 64 live voices in a session.

DirectMusic™ is an audio programming API designed to support musicians more than application developers, but has added capabilities that game or simulation developers may benefit from using. Some of these include:

- Multiple Audio Formats - DirectMusic™ supports multiple audio file formats, including MP3, wave and others
- DLS (Downloadable Sounds) - DLS is a standard for integrating several audio files for synchronized playback. While both OpenAL and DirectSound3D™ can shift frequencies, audio artifacts and distortions when the shift is far away from the original frequency will become apparent. DLS permits a programmer to use, for example, three audio files - one for low frequency sounds, one for intermediate frequency sounds and one for high frequency sounds, and then overlay them as necessary during playback. This precludes high- and low-pitch shifted artifacts and can produce much more realistic sounds in this type of scenario. Modeling vehicle engines provides an excellent example of the power of this feature. Three different audio files could represent low

RPM, intermediate RPM, and high RPM. Synchronized overlay and playback would preclude frequency-shifting artifacts and produce a smooth sounding engine.

- Separation of Buffers from Audio data - DirectMusic™ uses the OpenAL model of separating a source from its data. In DirectMusic™, a source (AudioPath) can be positioned, repositioned and manipulated in whatever fashion the developer desires. Audio data is then placed on the AudioPath when it is time to play.
- Audio Scripting - DirectMusic™ supports scripting long segments of sounds and permits the developer to introduce variability. This capability might enhance training simulations where ambient sound tracks could be developed and varied from run to run, without have to re-program audio sequences between each execution. Like OpenAL, DirectX™ (either DirectSound3D™ or DirectMusic™) is a full-feature audio programming API capable of delivering an exciting audio experience to the user.

While DirectSound3D™'s interface is the most difficult to learn and understand, it offers the most functionality of any audio API available. It is widely used in the video game industry, and has extensive support for programmers through Microsoft's MSDN library, web site and other gaming web sites.

D. EAX™ 3.0

EAX™ Audio Extensions is an audio API produced by Creative Technology to induce numerous types of audio effects, including reverberation, occlusion, obstruction, and exclusion (available at http://developer.creative.com). The goal of the API is to produce effects equivalent to modeling the acoustics of rooms, buildings, and other audio environments. It does this without the expensive CPU requirements of actually modeling geometry and audio ray tracing. EAX™ 3.0 is the current release version. EAX™ works as an extension to an underlying audio API - EAX™ is currently configured to work with both OpenAL and DirectX™.

A few of EAX™ 3.0 Audio Extension's capabilities include:

- Audio Environments - EAX™ supports both global and source-specific audio effects. Global effects are applied to the single listener in the environment, while source specific effects are independently applied to audio sources. EAX™ permits the application developer to select from over one hundred defined preset environments for global effects, and five

high level parameters to modify those presets to obtain the type of effect the developer desires. The EAX™ interface supports the ability to:

- Set or modify the environment or room size
- Set or modify the environment or room width or depth
- Set or modify the environment or room materials. For example, a hardwood floor will reflect a greater amount of sound energy than a carpeted floor and a wood wall will transmit more sound energy between rooms than a stonewall.
- Set or modify the environment or room height. A taller room will reverberate more than a shorter room.

- Audio Source Effects. As audio sources are moved in the environment, EAX™ can construct three main types of effects to simulate how those sources would interact with a listener:
  - Occlusion. Occlusion is the perceived effect of a listener and an audio source being in different rooms. Depending on the type of material separating the listener from the source (which is fully modifiable in EAX™), a variable amount of sound energy will penetrate the separator and arrive at the listener. EAX™ permits setting of various types of wall or room material.
  - Exclusion. Exclusion is the perceived effect a listener hears when the audio source is not in the same environment but is heard through a portal, such as a door or a window. Exclusion is an attenuation of reflected and reverberated sound energy while the direct path energy is relatively unaffected. EAX™ supports modification of parameters to achieve different effects for door or window size.
  - Obstruction. Obstruction refers to the perceived effect when the listener and the sound source are in the same environment, but an object is between the two. In this case, direct-path sound energy is attenuated, especially at high frequencies, and reflected and reverberated sound is left untreated.
- Multiple Audio Environments and Environmental Morphing. EAX 3.0 supports up to four simultaneous audio environments, permitting developers to acoustically model multiple rooms. Additionally, environmental morphing blends effects as listeners traverse portal between environments for a smooth audio effect.

Each of the three audio source effects is considered to be a “high level” parameter in the EAX™ API. Each effect sets or modifies a varied number of “low level” parameters in the API. The application developer has access to and can modify each of

the lower level parameters to fine-tune the implementation to achieve a specific audio effect.

EAX™ can be implemented in any audio application. Creative Labs has released a single API, called the EAX™ Unified Interface, which permits application programmers to write audio software without fear of what type of sound hardware is on the user's machines. The EAX™ Unified provides a single API for all versions of EAX™, and even will disable the effects if the host machine's sound card does not support EAX without terminating the application.

The most important issue to understand when using EAX™ is that it does not employ true architectural acoustics. It is strictly a numerical, parameter-based API for achieving certain effects. Experience suggests that numerous iterations of parameter tweaking are required to achieve integrated and synchronous effects for an interactive application where the listener and sources are in constant movement.

E. SOFTWARE API SUMMARY

In conclusion, both OpenAL and DirectX™ are extremely powerful audio API's capable of delivering a rich audio experience. When combined with EAX™, either can come quite close to simulating an audio experience that rivals reality. DirectX™ module API's are much more difficult to understand and comprehend, and result in a very steep learning curve for beginning programmers. For beginning programmers, OpenAL provides an easy to understand API that can familiarize one with the basic of audio programming yet still create a dynamic audio environment. If the application requires live voice, DirectX™'s DirectSound3D™, DirectPlay™, and DirectVoice™ offer the only integrated API available. However, latency in DirectVoice™'s live voice stream may be too great for some applications.

F. AUDIO RESOURCE MANAGEMENT

Audio resource management refers to the techniques employed to maximize the capabilities of audio hardware while meeting the audio needs of the application. Any piece of audio hardware, from a computer sound card to high-end audio equipment, faces

limitations as to how many sound sources it can play simultaneously and what types of audio processing it can accomplish. For purposes of this thesis, audio resource management will only address PC sound cards, as the audio architecture presented here is centered on the PC as the delivery platform for the virtual environment.

All sound cards produced today, including Creative Technology's Audigy™ sound card, contain a fixed number of sound buffers for processing audio data. Audio data can be either statically loaded into a sound card buffer in its entirety or continuously streamed into the buffer. In either case, the number of buffers on the card will limit the number of sounds available for simultaneous playback. Audigy™, for example, contains sixty-four sound buffers. Thirty-two buffers exist for 3D processing and thirty-two buffers exist for 2D (stereo) processing. This means that up to sixty-four non-3D sounds may be played simultaneously or thirty-two 3D sounds may be played simultaneously (a hardware accelerated 3D sound buffer can always play 2D sounds.) Now, sixty-four sounds may appear as a large number, but in actuality it is not, especially in complex audio environments such as CQB. A scheme must be developed to manage which sounds are played and, in the event that an attempt to play more sounds than the number of sound buffers, determine a priority rule for selectively picking which are the "most important" sounds to be played. DirectSound3D™ incorporates a technology known as Voice Management. Voice Management automatically prioritizes which sounds are to be played based on criteria set forth by the programmer. In general, the criterion used, whether through Voice Management or by manual programmer control, is to select the sounds closest to the listener. In this manner, if a virtual environment had more sounds than the PC sound card could support, sounds the most furthest away from the listener would be deselected for playback until such a time as the listener moves closer.

Another issue with audio resource management is how effects are processed. Certain types of effects can be processed in the main CPU, such as volume changes, pitch or frequency changes, and distance attenuation. While these types of effects, if produced in software, are possible, software processing of audio data is highly expensive and induces latency in the overall system. If all audio effects are processed on the sound card, no CPU degradation due to expensive audio processing will occur. Additionally,

some audio effects, such as those produced by EAX™, can only be created in hardware buffers on the sound card.

Although the technology of today's PC sound cards is rapidly evolving, it still places a burden on the programmer to manage audio resources. Programmers may elect to use an automatic resource management tool such as Voice Management, or develop their own resource management technique.

G. AUSIM3D GOLDSERVER

Due to the described latency in live voice by any VoIP application or client, including DirectVoice, hardware implementations of live streaming voice were examined for alternatives to a VoIP application or component client. The criteria for selection of a hardware alternative consisted of the same criteria mentioned previously: utilize COTS technology whenever possible; attempt to minimize cost; and ensure the system would be compatible with a PC-based virtual environment training system. Additionally, any hardware implementation must be:

- Compatible with any or all of the API possible solutions discussed in this chapter
- Capable of spatializing the voice signal in three dimensions. Live voice in a virtual environment should act like sounds in a virtual environment, emanating from point sources, directionalized and spatialized, and perhaps most importantly, movable within the virtual environment. Voices should appear to be coming from avatar representations of the players in the shared virtual environment.

The search for an alternative hardware implementation was brief; there were only two hardware solutions to live streaming voice currently available. One solution was to connect separate virtual environments training systems with a simple telephone connection for live voice. While fairly easy to implement - telephones with headset jacks could be routed to each location in a shared virtual environment for live voice communications between participants. However, a telephone-type communications system would be unable to spatialize the voices to co-locate them with their avatar "owners". The only remaining solution found was the Ausim3D GoldServe Audio Localizing Server System, produced by Ausim3D of Los Altos, California (http://www.ausim3d.com).

The GoldServe is capable spatializing live voice with an advertised latency of 5 – 10 ms, virtually imperceptible to the participants. Characteristics such as distance attenuation, rolloff, and source directivity are additional capabilities of the Ausim3D GoldServe.

There are two main limitations to the GoldServe when implementing it as the live streaming voice solution. First, the GoldServe only supports headphone audio. Virtual environment audio delivery systems range from loudspeakers to headphones to combinations of both. Research at the Naval Postgraduate School by Scorgie and Sanders (2002) suggest there is a relationship between the sense of presence and immersion in the virtual environment and the selection of which type of audio delivery system. Their initial findings indicate that loudspeakers are superior to headphones in providing the user of the virtual environment with a sense of presence and immersion. Further study is warranted, but a virtual environment capable of delivering audio only through headphones may face a limitation on its immersive capability. They also examined whether an audio delivery system combining a subwoofer with headphones had any effect on the sense of presence. Their findings suggested no effect on presence for this treatment.

Second, the GoldServe Audio Localizing System requires the co-location of the individual virtual environment trainers or simulators. The GoldServe, while network capable for accepting audio commands and data, does not support an IP-based audio delivery system. The GoldServe, as part of a multiple participant virtual environment, requires that each of the participants be directly connected to the GoldServe hardware through standard headphone cables and headsets. This limitation precludes an implementation for a scenario where the multiple participants are distributed across a wide area or large network. Additionally, the GoldServe, depending on its configuration, will only support a fixed number of participants. The GoldServe system used as part of the development of the audio architecture for this thesis is limited to four active listeners. It is a single processor system. Individual GoldServe systems can be upgraded to dual processors, in which case the number of listeners double to eight. Additionally, chains of GoldServes can be constructed to permit creation of large numbers of listeners necessary for any particular application. For example, four dual processor machines could support

up to 32 listeners. Naturally, purchasing a dual processor Ausim3D system or multiple systems will provide a greater number of listeners (users), but the number will still remain fixed.

Even in light of these limitations, for those virtual environments involving multiple, physically co-located participants, the GoldServe is the best solution for providing live streaming voice available. Considering that many military virtual environment training systems are being designed to deploy onboard ship for individual or team training, the GoldServe is an extremely viable solution. Onboard ship, teams of virtual environment users will likely be co-located with potentially severe space limitations. It is likely that each simulator will not be able to capture enough space for setup of a loudspeaker audio delivery system considering the limited space requirements. Headphone delivery systems may be the only solution to deployable virtual environment training systems. Additionally, the physical co-location of the users matches the GoldServe limitation on distribution of users. For other implementations, where participants are distributed across a wide area or network, VoIP remains as the only viable solution for providing live voice.

Future upgrades to the Ausim3D GoldServe will implement a broad range of room acoustic modeling capabilities. When these upgrades are complete, the GoldServe, combining spatialized, live voice with a run-time room acoustics processing capability, will be the premier audio system for developing fully interactive, multi-participant virtual environment training systems and simulations.

H. OVERALL SYSTEM ARCHITECTURE

The audio architecture described in this section is meant to cover many different possible implementations in virtual environments and is a general overview. Specific implementation for each of the software components is found in Chapter IV. It can be described as a "tool box" of hardware and software components that can be arranged and combined in different ways for different virtual environment training systems, including:

- Single independent user
- Multiple physically co-located users
- Multiple distributed users

The audio architecture is subdivided into two sections, a core component found on all implementations, and a live voice component, from which the virtual environment training system developer will select one of two options for implementation.

The core component of the audio architecture is a combination of DirectSound3D™ and EAX™ software implemented on each system. Other than live voice, each system, whether connected to others or not, is responsible for generating all sounds and sound effects heard by the user of that system. This core component supports all three implementation options described above, and is in compliance with the requirement that the audio architecture be COTS technology, low cost, and PC-based. Virtual environment audio is generated locally through the PC sound card. Even when connected to other virtual environment participants, sounds for remote users are generated locally by exchanging network packets with pre-formatted audio data. Each user's system will contain a library of necessary audio files for playback of all sounds potentially heard in the virtual environment. The core component is a software-only API designed to be compatible with a standard PC. Specific implementation details are found in the next chapter.

For single, independent users, only the core component of the audio architecture is implemented as shown in Figure 8. The core component can meet all of the audio requirements for a single-user virtual environment.

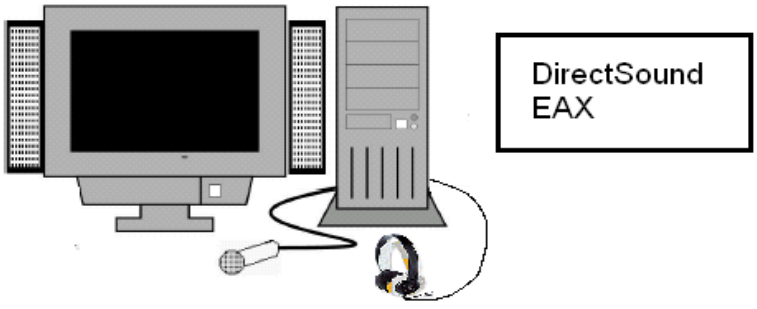


Figure 8. Single Independent User Audio Implementation.

For multiple, physically co-located users, the audio architecture will contain the core component and the Ausim3D GoldServe (Figure 9). In this case, DirectSound3D™ and EAX™ will provide all sounds and sound effects for each user on his or her individual system. An IP network connection between users will provide a data path for transmission of audio data relating to remote participant sounds and actions that generate sounds. Audio generated by each individual system's sound card will be routed and connected to the GoldServe as a live audio feed. Since spatialization of audio for sounds and sound effects has already been processed on the individual user's systems, there is not requirement for additional processing on the GoldServe, other than a simple stereo separation of audio channels from the individual PC systems. Each user's system is connected via two standard ¼" audio cables, left and right audio channels from the individual system sound card, to the GoldServe and the channels are panned left and right respectfully. Each user's microphone is also connected as a live voice input to the GoldServe, and spatialization processing is completed by the GoldServe to match the situation in the virtual environment. A separate PC system connected to the IP network will function as an audio server, connected to the GoldServe via a RS-232 data cable to provide the GoldServe with positioning information for all users in the shared virtual environment. The server may be one of the participants in the shared virtual environment, or another PC acting solely as a central connection point between the IP network of virtual environment users and the GoldServe. When possible, it is recommended that a stand-alone system be employed as the audio server for the GoldServe. Virtual environment graphics can be very CPU intensive to process, and requiring one of the user systems to additionally serve as the audio connection to the GoldServe may induce processing latency in the display.

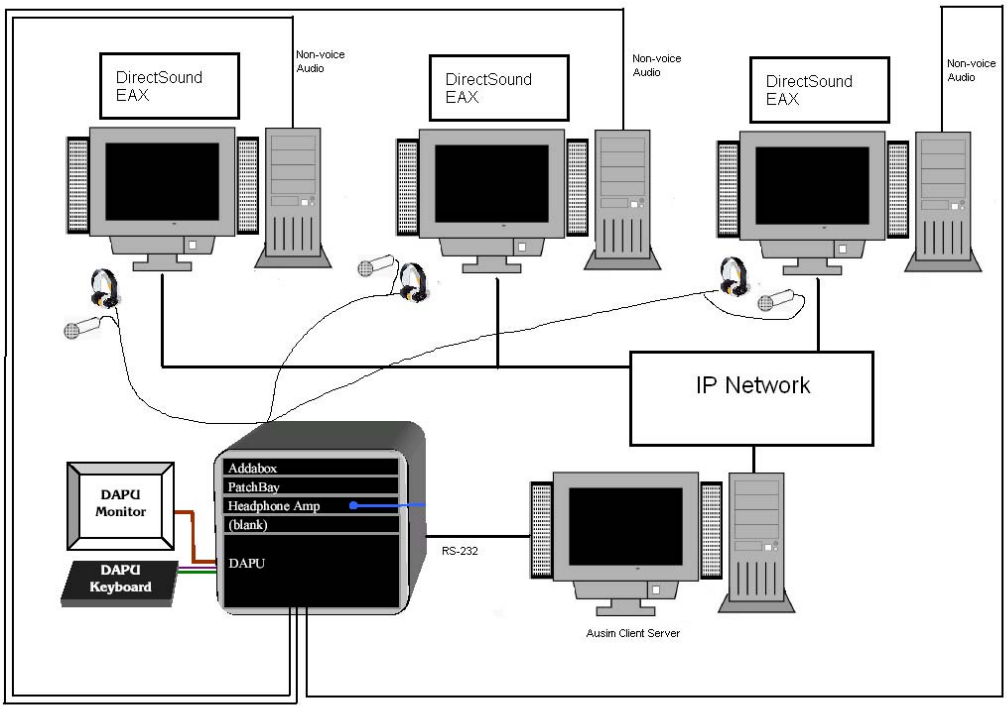


Figure 9. Multiple, Physically Co-Located User Audio Implementation.

When multiple users are not physically co-located and distributed over a wide area, the audio architecture will contain the core component and use DirectVoice™ for live streaming voice (Figure 10). In this instance, no audio server is required for connection to any additional hardware. DirectVoice™, using DirectPlay™ as its networking layer beneath streaming voice signal, is established either in conjunction with a data network for the simulation or as a stand-alone connection only for the voice stream. The DirectPlay™ network should be established as a peer-to-peer connection, meaning that each participant in the network transmits data to each of the other participants. DirectPlay™ and DirectVoice™ support up to sixty-four simultaneous users in a shared virtual environment.

The benefit of this architecture is that the voice stream, DirectSound3D™ streaming sound buffers reserved for live voice signals, can be processed by EAX™ for sound effects like other sounds in the environment. Occlusion, obstruction, exclusion and reverberation are effects that can be applied to voice streams just as they can be applied to other sounds in the environment. The principle limitation of this architecture

is the latency in the voice signal, as discussed. Chapter V outlines an experiment conducted to measure and compare the latency in live streaming voice between the GoldServe and DirectVoice™ under various network-loading conditions. If multiple users of a shared virtual environment must be distributed and cannot be co-located, VoIP currently offers the only solution for live streaming voice. Training system designers must address whether the latency in the live voice stream is acceptable or unacceptable. If unacceptable, alternatives, such as text-based messaging, can be implemented for real-time communications between users.

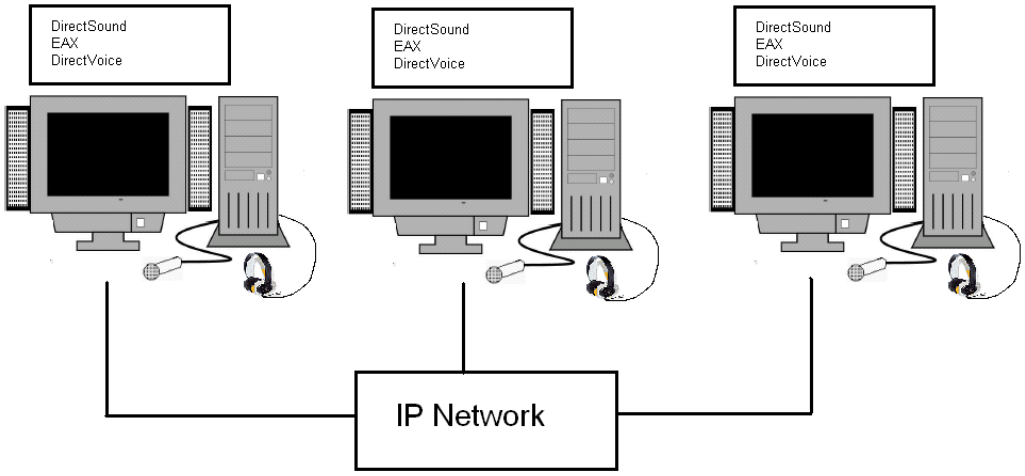


Figure 10. Multiple Distributed User Audio Implementation.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. VOICE LATENCY ANALYSIS

A. INTRODUCTION

Latency has been shown to be a critical issue in live voice in virtual environments. The latency in live voice can arise from several areas, including network latency and audio compression.

Network latency is induced when the IP-based network is heavily burdened with transmitting a high amount of data packets. The level of network traffic will vary depending on network configuration. Depending on virtual environment network design, network latency can be extremely severe and significantly add to latency in both data transmission and the live voice signal

Latency in live voice can also be induced at the source through a process known as compression. Live voice, like any live audio input, is generally sampled at a known rate and digitized into text or binary data for transmission across the network in standardized data packets. Numerous compression schemes have been created to lower the sampling rate, package more data into each voice packet for transmission over the network, and decode the network voice packets at the receiver. Compression, while attempting to reduce the number of voice data packets to limit network loading, induces latency by storing data about the voice signal over a small period of time before packaging and transmission. Depending on the selected compression algorithm, this small amount of time can be very noticeable to live voice users in a virtual environment, especially when the virtual environment displays a graphical image for users to compare with the voice signal. Without a graphical display, users would not easily determine that the voice signal delay was latency as opposed to human delays in speaking. However, in virtual environments where the graphical display and the audio display must be highly synchronized, latency in voice can be very obvious. At best, the latency can be distracting; at worst, it can significantly decrease immersion and training effectiveness.

B. EXPERIMENTAL DESIGN

1. Apparatus

To control the type and quality of the audio signal being used for testing, both between the Ausdim3D GoldServe and DirectVoice™ and within each run for each implementation, the decision was made not to test an actual live voice. Instead, a 1000Hz tone would be generated and pulsed through each system. The 1000Hz frequency was selected as it falls within the normal voice range of frequencies, from 500 Hz to 2000Hz. To generate the pulse, a Hewlett Packard 8015A Pulse Generator was employed. The 8015A Pulse Generator is capable of delivering single, burst, or continuous pulses of audio from one nanosecond to one hertz.

To view the output of each implementation's test runs, a Tektronix TDS 3012 Digital Phosphor Oscilloscope was utilized. With the capability to simultaneously display two signals (pulse and resulting audio signal from receiver), the oscilloscope could provide a graphic display of the original pulse and the resulting received signal. The signal from the pulse generator was routed to the oscilloscope.

2. Procedures

The Ausim3D GoldServe has its own internal latency measurement software, but this option was not utilized. The testing software only measures the latency of the signal during processing on its own sound card. While this is the bulk of latency on the GoldServe, it cannot be considered as a complete measurement as it does not take into account latency in the signal path through other hardware components.

For the Ausim3D GoldServe, a simple test application was developed to create two listeners, each with a live voice channel. For the test, the output of the pulse generator was connected to the microphone (live voice) input and the headphone output was connected to the oscilloscope for display. Twenty-three individual runs were completed, each with a single audio pulse. Individual run graphic outputs were saved and are shown in Appendix C.

Early in the test, an attempt was made to vary the listener positions to determine if it had an effect on latency of the voice signal. It was quickly determined that the GoldServe implements an algorithm to calculate time of arrival using the standard speed of sound to delay audio based on distance from virtual source to virtual receiver. This capability was under development by Ausim3D, but not specifically listed in the product documentation. DirectVoice™ does not support time of arrival delay in its audio processing. Therefore, to ensure that no additional latency in the received signal was due to the GoldServe's distance delay algorithm (which is not latency but an attempt to reproduce sounds more realistically by modeling time of arrival and distance delay), positions were varied in three dimensions but limited to no more than one meter of virtual separation.

An equivalent procedure was employed to test DirectVoice™ latency. In this case, the pulse generator was connected to one PC's sound card microphone input and the oscilloscope was connected to the headphone output of a second PC. A DirectVoice™ connection was made between the two PCs, and thirty measurements were taken. Graphic results can be seen in Appendix C.

C. RESULTS AND ANALYSIS

The numerical results of the experiment are summarized in Table 2.

Treatment	Average Latency	Standard Deviation
VoIP – DirectVoice	205	15.2
Hardware – GoldServe	13.1	0.2

Table 1. Average Latency Measurement in Milliseconds.

The results indicate a large difference in the latency of the GoldServe's live voice implementation compared to DirectVoice™'s implementation. Two points must be made before the analysis. First, the GoldServe is not subject to network delays or heavy message loading. As seen back in Figure 9, the network connection for transmitting

virtual environment state data is not even connected to the GoldServe. The GoldServe is connected to the network via a "repeater" node - a computer not serving as one of the participants in the simulation. Therefore, factors that could affect network performance will not affect the GoldServe audio processing or the live voice latency. The measured GoldServe latency would stay relatively constant, regardless of other requirements placed on the virtual environment network. Second, the DirectVoice™ VoIP implementation will be affected by network performance. If the state information necessary to maintain the virtual environment burdens the network with heavy message flow, either voice latency will increase or voice quality will degrade as increasing numbers of voice data will be lost or reconstructed out of sequence.

In other words, the average latency measurements must be viewed differently. The GoldServe average latency, 13.1 ms, can be seen as an average under virtually any network loading condition. The DirectVoice™ latency average, 205 ms, should be seen as a best-case scenario. The scenario is best case in that the DirectVoice™ connection was only between two computers, and no burden was placed on the network connection to support a significant level of virtual environment state information transmission. If the numbers of participants were to grow or the level of network transmissions were to increase, either as a result of more participants or a requirement to transmit additional state information, the latency would increase.

Returning to the discussion of synchronization in a virtual environment, Bolot and Fosse-Parisis (1998) determined that a de-synchronization between the audio and visual presentations in a virtual environment should not exceed approximately 185 ms. If de-synchronization is greater than 185 ms, it will become noticeable to the user and distracting from the environment. In a military virtual environment training system, where the subject matter being trained in the virtual environment may crucial to real world life and death situations. If the users are distracted during the training due to latency in voice communications, it may have a detrimental effect on the battlefield. The GoldServe's latency measurements fall far below this threshold. The VoIP measurements, remembering that they are best case, exceed the threshold, and will likely exceed it further as the training system becomes more complex with more participants.

The GoldServe solution to live voice is vastly superior to a VoIP live voice implementation.

D. SUMMARY

From the results and analysis, it is shown that a VoIP implementation of live voice in a virtual environment will have a significantly higher level of latency than a hardware live voice implementation like that found in the Ausim3D GoldServe. Voice latency greater than 185 ms will not only be noticeable to users, but potentially distracting and detrimental to training. The GoldServe's superior latency measurements indicate it is the best solution for live voice. Only in those situations where the networked virtual environment participants cannot be co-located should VoIP be utilized for live voice. It is speculated that whether an application requiring live voice will suffer from some level of training degradation due to voice latency is highly application specific. For example, the extremely complex CQB environment described by Greenwald (2002) requires live voice in virtually every phase of operations. CQB, by its very nature, is an extremely rapid and volatile activity. Verbal communications between team members is quick, precise, and demands immediate response. If a virtual environment training system employs a high-latency live voice sub-system, the latency may not only degrade the verbal communications, but also contribute to operational mistakes of a potentially life-threatening nature. In the real world, military members executing a CQB mission would never accept a radio communications system that induced potentially deadly delays in critical communications. However, other training activities may not suffer at all. For example, a virtual ship-handling environment designed to simulate two or more vessels working together may not seriously suffer from latency in simulated radio communications. The need for low latency live voice has been demonstrated, but is not always applicable for every training system. Determining which types of applications require low latency live voice and those that can accept a higher level of latency bears further analysis.

THIS PAGE INTENTIONALLY LEFT BLANK

V. SOFTWARE IMPLEMENTATION

A. INTRODUCTION

This chapter provides a specific implementation in software for two APIs. The first, gfAudio, combines DirectSound3D™ and EAX™ into s single API capable of a fully immersive auditory virtual environment. The second, auServerLib, is an API capable of accepting an external network connection to a multi-user, shared virtual environment and communicating with the Ausim3D GoldServe hardware for live voice. Both were developed to work in conjunction with gfLib, a virtual environment-authoring library developed at the Naval Postgraduate School by Erik Johnson. Due to licensing requirements for other virtual environment authoring libraries, such as Vega from MultiGen Paradigm and Unreal from Epic, gfLib provided a high-quality, open-source graphics library that could be used to test the implementation of both gfAudio and AuServerLib. Although there are specific methods, variables and functions that permit gfAudio to efficiently integrate into a gfLib application, with little modification gfAudio can be separated into a stand alone API for use with any virtual environment authoring library. AuServerLib is constructed to work with any type of networked virtual environment. Both libraries were constructed using standard object-oriented programming design principles.

AuServerLib was constructed to be more than simply a "connection" protocol for external networked virtual environments to communicate with the Ausim3D GoldServe. It is designed to be a fully functional API capable of exploiting all functionalities of the GoldServe, and can be used to construct many different types of applications. The auServerLib API can be used for such applications as audio or voice testing, or single- or multiple-user acoustic environments. Although in the architecture described for virtual environments in this thesis only utilizes the GoldServe for live spatialized voice, the auServerLib API is capable of integrating pre-recorded wav files and live input channels into any virtual environment.

In the next two sections, each of the libraries main source code and implementation details is examined. The core functions and architecture are described as

to how each class accomplishes it role within the libraries. Appendices A through D provide complete documentation and source code for further reference. The following typeset convections are used:

- Courier new text will indicate source code. For example, a portion of source code contains: `EAXDirectSoundCreate`
- Bold text will be used for software objects created as interfaces to either DirectSound3D™ or EAX™ internal source code or methods. For example, an object will appear as: **IDirectSound**

B. GFAUDIO

1. gfAudioGlobal

The gfAudioGlobal.h and gfAudioGlobal.cpp files contain methods to initialize both DirectSound3D™ and EAX™. Initialization is accomplished as depicted in Figure 10:

```
if( FAILED( hr = EAXDirectSoundCreate( NULL, &pDS, NULL ) ) )

. . .
if( FAILED( hr = pDS->SetCooperativeLevel(
       (HWND)win->GetGZWindow()->getHandle(), DSSCL_PRIORITY ) ) )
```

Figure 11. DirectSound3D™ and EAX™ Initialization Code.

The `EAXDirectSoundCreate` method creates an interface object of type **IDirectSound**, which is the primary interface in DirectSound3D™ with the host machine's sound card. The **IDirectSound** object is used to create all other interfaces in DirectSound3D™. Setting the cooperative level to `DSSCL_PRIORITY` permits DirectSound3D™ to share resources on the sound card with other applications running on the host machine. Like other windows-based applications, DirectSound3D™ uses a "window focus" design where sounds associated with a particular application window will only be heard through the sound card when the application window has the focus on the desktop.

Once the IDirectSound3D™ interface is constructed, the primary buffer for all sounds is created:

```
if(FAILED( hr = pDS->CreateSoundBuffer(&pBufDesc, &pDSBPrimary,
       NULL)))
. . .
```

Figure 12. Creating the Primary Sound Buffer.

The primary sound buffer, `pDSBPrimary`, can be thought of as the final mixer and combiner of all sounds playing simultaneously in the audio environment. The primary buffer format can be set to accept pre-recorded sounds with different sampling rates; generally accepted sampling rates are either 22050 Hz or 44100 Hz. Most commercially available recording software packages contain options for recording at these and other sampling rates.

The gfAudioGlobal files contain a sub-class named gfAudioNet, which is designed to play sounds in a networked, shared virtual environment. For example, in a two-person shared virtual world, when either participant initiates an action that produces a sound, the other participant should be able to hear that sound as well, taking into consideration such conditions as distance between users in the environment and the presence of obstacles or obstructions. When a developer determines that a sound should be able to be heard between users or participants, gfAudioNet automatically receives the notification to play a remote user's sounds on a local user's machine. It is assumed that in a networked, multiple user virtual environment, each user has access to the same library of stored, pre-recorded wav files. Beyond the presence of the pre-recorded wav file, individual user's applications do not have to be configured for every time of sound that other user's may initiate. The gfAudioNet class automatically discovers the name of the remote sound, creates the necessary gfSoundObject associated with the sound, and plays it according to specifications. Once created, these sound objects are stored for later use.

2. gfListener

The gfListener class is responsible for acting as the ears of the observer in the virtual environment. As in the real world, the listener's location and orientation are dynamic and moveable. The portion of the source code responsible for listener object creation is shown in Figure 12. After gfAudioGlobal initializes DirectSound3D™ and

creates the primary buffer, the primary buffer is used to create and obtain an interface to the DirectSound3D™ **IDirectSound3DListener** object, which is a software interface to permit setting of listener properties that affect all sounds and the global audio environment.

```
if(FAILED( hr = pDSBPrimary->QueryInterface(IID_IDirectSound3DListener,
       (void**)&pDSListener)))
. . .
if(FAILED( hr = pDS->CreateSoundBuffer(&sndBufferDesc, &pDSB, NULL)))
. . .
if(FAILED(hr = pDSB->QueryInterface(IID_IDirectSound3DBuffer,
       (void**)&pDSB3D)))
. . .
if(FAILED( hr = pDSB3D->QueryInterface(IID_IKsPropertySet,
       (void**)&pEAXListener)))
. . .
```

Figure 13. gfListener Configuration Source Code.

The **IDirectSound3Dlistener** object, `pDSListener,` is essentially software "ears" of the user in the virtual environment. Methods in the gfListener class permit setting the location, orientation and elevation of the listener. Orientation is particularly important in that it allows for 3D spatialization of sounds in the environment to occur. For example, if the listener's head orientation changes, such as rotating the head from looking forward to looking to the left, sounds originally emanating from forward of the listener will not appear to the right.

Figure 12 also shows the creation of other DirectSound3D™ objects, including creating a sound buffer, creating a 3D sound buffer and creating an interface to an EAX™ property set, `pEAXListener.` The sound buffer and 3D sound buffer are only created to obtain the interface to the EAX™ property set, and then discarded. EAX™ property sets are software objects and interfaces used to set EAX™ values. They are standardized methods using standardized variables for creating such effects as environmental reverberation. EAX™ property sets are created for the listener and each individual sound object in the environment. Effects set through the listener's EAX™ property set are global in nature; they apply to all sounds in the environment. It is analogous to applying a sound effect filter in our ears - every sound arriving at our ears is

filtered for the given effect. An example of setting a global environmental effect, such as the listener being positioned inside a sewer pipe, is shown in Figure 13. EAX™ contains over 100 preset acoustic environments for software developers to use to create acoustic effects in virtual environments. EAX™ 3.0 has expanded the number of presets to over one hundred different acoustic environments.

```
EAXLISTENERPROPERTIES env = EAX30_PRESET_SEWERPIPE;
if(FAILED( hr = pEAXListener->Set(DSPROPSETID_EAX_ListenerProperties,
       DSPROPERTY_EAXLISTENER_ALLPARAMETERS, NULL, 0, &env,
       sizeof(EAXLISTENERPROPERTIES))))
. . .
```

Figure 14. Setting an EAX™ Property for the Listener.

The gfListener class contains other methods to permit the following:

- Tethering the listener to an object. The listener can be "tied" to an object in the virtual environment for automatic position and orientation updating. In most cases, the object to tether a listener to is the observer in the environment, which represents the user. In most virtual environment development libraries, including gfLib and Vega, an observer is considered to be the "eyes" of the user. Tethering the ears to the eyes of the participant is naturally expected.
- Doppler Effect settings. For a natural Doppler effect, velocities of the listener and the sounds in the environment must be known. DirectSound3D™ does not automatically calculate velocity by measuring change in position; the developer must manually set the velocity. A simple algorithm to measure distance change over time determines velocity. Additionally, the Doppler Effect can be exaggerated. In many instances, a developer may want to exaggerate the Doppler Effect to make it more noticeable to the user. The gfListener class contains a method to set a factor that either over- or under-exaggerates the Doppler effect, based on developer preference and design.
- Rolloff and distance calculations. In addition to head orientation, distance from source to "ears" is the other primary characteristic by which we discern spatialization of sounds. The gfListener class permits a developer to either over- or under-exaggerate the global attenuation of sounds as they move further away from the listener. In certain virtual environment scenarios, the exaggeration may be required to provide the desired attenuation "feel" for distance and near sound sources.

3. gfSoundObject

The gfSoundObject class represents a sound in the virtual environment. This class has the capability to load, locate, play, and loop pre-recorded audio files using the wav audio file format. Additionally, gfSoundObject contains methods to set sound-specific effects, such as occlusion, obstruction and exclusion. These effects are rendered through the integration of EAX™ with DirectSound3D™. EAX™, to create these types of effects, just obtain hardware resources on the host machine's sound card. EAX™ processing is only permitted in hardware; thus, if the host machine's sound card does not contain the necessary sound buffers for EAX™ processing, EAX™ effects will not be processed nor heard. Most sound cards produced today, such as Creative Labs Audigy and SoundBlaster series audio cards, contain between 16 and 64 hardware buffers on the sound card itself, permitting EAX™ processing for an equivalent number of sounds in the environment. However, limiting a virtual environment to between 16 and 64 sounds significantly limits the environment's ability to provide a rich and fully immersive audio environment. The gfSoundObject class overcomes this limitation by dynamically obtain sound card hardware resources only when a sound is playing or looping. Upon completion of the play or loop sequence, the buffers on the sound card are released and resources are returned to the operating system. In this manner, the only limit on the number of sounds in the virtual environment is the number of simultaneously playing sounds, still limited to between 16 and 64, depending on sound card hardware resources. However, an unlimited number of gfSoundObjects can be created and stored for playback.

The gfSoundObject, unlike the gfListener, does not create the necessary DirectSound3D™ and EAX™ software objects at instantiation. Instead, the DirectSound3D™ and EAX™ interfaces are created only through a method call to play or loop a sound. Obtaining the necessary hardware resources to play or loop a sound are is show in Figure 14.

```
if(FAILED( hr = pDS->CreateSoundBuffer(&m_DSBufDesc, &pDSB, NULL)))
. . .
if(mType == GF_3D)
{
```

```
if(FAILED(hr = pDSB->QueryInterface(IID_IDirectSound3DBuffer, (void **)
       &pDSB3D)))
. . .
if(FAILED(hr = pDSB3D->QueryInterface(IID_IKsPropertySet,
       void**)&pEAXSource)))
. . .
if(FAILED(hr = pEAXSource->QuerySupport(DSPROPSETID_EAX_BufferProperties,
       DSPROPERTY_EAXBUFFER_ALLPARAMETERS, &support)))
. . .
if ( (support & (KSPROPERTY_SUPPORT_GET | KSPROPERTY_SUPPORT_SET)) !=
       (KSPROPERTY_SUPPORT_GET | KSPROPERTY_SUPPORT_SET))
. . .
```

Figure 15. gfSoundObject Obtaining Resources for Play or Loop.

First, a DirectSound3D™ hardware buffer, `pDSB` is created using the **IDirectSound** object created in gfAudioGlobal. The `pDSB` object is a DirectSound3D™ **IDirectSoundBuffer** object that acts as the final mixer for any effects, such as 3D spatialization, EAX™ effects, or pitch and volume changes for the sound. To permit spatialization, an IDirectSound3Dbuffer is obtained, `pDSB3D`. Like the gfListener, an EAX™ property set, `pEAXSource`, is created for the occlusion, obstruction, and exclusion effects processing. Unlike the gfListener, EAX™ effects processed by an individual gfSoundObject are only for that sound; the effect is considered to be source specific.

When it comes time to play the sound in the virtual environment, the process of obtaining resources, setting parameters and effects, and locating the sound in 3D is accomplished as depicted in Figure 15.

```
if(!loop) ReleaseResources();
. . .
if(!ObtainResources())
{
       CheckPlayingSounds();
       if(!ObtainResources()) return;
}
if(pDSB )
{
       if(mType == GF_2D )
       {
              SetPan(mPan);
       }
       if(pDSB3D)
       {
              pDSB3D->SetMinDistance(mMinDistance, DS3D_DEFERRED  );
              pDSB3D->SetMaxDistance(mMaxDistance, DS3D_DEFERRED  );
```

```
        pDSB3D->SetConeOrientation(mConeDirection[0], mConeDirection[1],
               mConeDirection[2], DS3D_DEFERRED );
        pDSB3D->SetConeAngles(mMinConeAngle, mMaxConeAngle,
               DS3D_DEFERRED );
        pDSB3D->SetConeOutsideVolume( (long)mOuterConeVolume,
               DS3D_DEFERRED );
        if(mRel)
        {
               pDSB3D->SetMode(DS3DMODE_HEADRELATIVE, DS3D_DEFERRED  );
        }
        else
        {
               pDSB3D->SetMode(DS3DMODE_NORMAL, DS3D_DEFERRED  );
               pDSB3D->SetPosition( mPosition->X(), mPosition->Y(),
                      mPosition->Z(), DS3D_DEFERRED  );
               SetOcclusionSettings();
               SetExclusionSettings();
               SetObstructionSettings();
               if(g_gfListener) g_gfListener->CommitDeferredSettings();
        }
}
if (loop)
{
       pDSB->Play(0, 0, DSBPLAY_LOOPING);
       if(isNetworked) Send(GF_LOOP);
}
else
{
       pDSB->Play(0, 0, 0);
       if(isNetworked) Send(GF_PLAY);
}
SetPitch(mPitch);
SetVolume(mVolume);
```

Figure 16. Playing a Sound with gfSoundObject.

When a sound is played, the gfSoundObject first determines whether the desire to play the sound one time or to continuously loop the sound. If the sound is to be played once, any resources obtained for a previous play are released. This precludes multiple hardware buffers being obtained from the sound card for a single sound. This is not required for a looping sound. Since looping sounds will play continuously until an overt action terminates the sound, the termination will release the resources upon stopping the looping sound. For a single play, there is not overt method call or action that terminates play; thus, for single-playing sounds, the release of previously held resources ensures optimum usage of precious hardware resources on the sound card. The gfSoundObject then attempts to obtain the necessary hardware resources. If it is successful, it continues through the remaining stages of the method. If not, the gfSoundObject, through the `CheckPlayingSounds()` method, runs through a global list of all sound objects to determine if any gfSoundObject that currently is holding resources on the sound card has

completed playing. If so, it releases those resources, returning those sound card resources to the operating system for usage elsewhere. Upon completion of the `CheckPlayingSounds()` method, the gfSoundObject once again attempts to obtain hardware buffers on the sound card. If successful on this second attempt, processing continues. If not, the method is aborted and the sound is not played. In this case, the decision as to how to handle playing the respective sound is left up to the application developer - this case can only arise when the developer is attempting to play more sounds than the sound card hardware will support.

The gfSoundObject class contains the functionality to permit directivity of a sound. Directivity refers to sounds that emit energy in a specific direction instead of emitting energy omni directionally. Figure 16 shows a directional sound source. The volume of the inner cone and outer cones and direction of sound emanation are controllable by the developer. The volume of the sound source in the transition zone is automatically calculated by DirectSound3D™, attenuating from the level in the inner cone to the outer cone linearly.

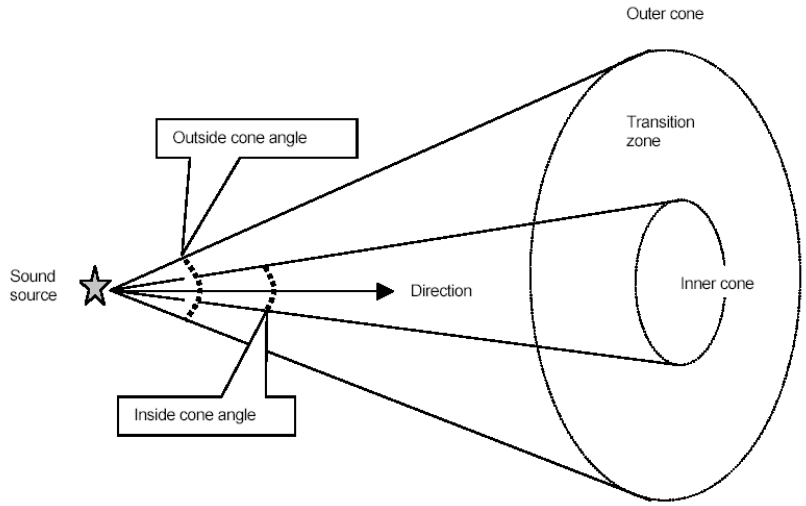


Figure 17. Directional Sound Source.

Processing continues by setting all of the stored characteristics of the sound - position, directivity (cone), minimum and maximum distance, and occlusion, obstruction or exclusion effects.

The gfSoundObject class allows for two general types of sounds in the environment - 2D or 3D sounds. 2D sounds are those that are not spatialized in three dimensions. Sounds that are 2D are always rendered as if they are located at the listener's position. These sounds can only be panned left or right. Panning refers to separating the left and right stereo channels and applying individual gain (volume settings) to each to mimic positioning the sounds to the left or the right of the listener. On face value this would not seem beneficial to using 3D sounds, which permit full spatialization around the listener. However, most sound cards contain separate hardware resources and buffers for processing 2D sounds, which are distinct and separate from those resources for processing 3D sounds. For sounds associated with the listener that will always be positioning relative to the listener, such as footsteps, utilizing a 2D gfSoundObject will free 3D hardware resources for utilization by other sounds. In this manner, applications may increase the number of simultaneously playing sounds by utilizing a mixture of 2D and 3D sounds, optimizing the hardware resources on the sound card.

Other methods in the gfSoundObject class do the following:

- Set pitch of the sound. Three methods permit manual frequency changes for the sound: increase pitch, decrease pitch, and set pitch. Changing the pitch changes the sampling frequency of the stored wav file during playback. This is very useful in simulating sounds such as engine noise that changes with an RPM change.
- Setting volume. Three methods permit volume manipulation of the sound. All volume changes are applied to the original gain (volume) of the sound. The volume cannot be increased over the original recorded volume, but can decrease the volume of the sound. If the volume is decreased, it can be increased, but only to a maximum of its original recorded level.
- EAX™ effects. The three source-specific EAX™ effects (obstruction, occlusion and exclusion) can be set for each individual sound. The gfSoundObject class only permits manual setting of these effects; for automatic updating of effects as the listener moves through a virtual environment, see the documentation regarding gfAudioEnvironment, gfAudioEnvironmentManager, and gfAudioEnvironmentTransition.
- Tethering a sound to an object. The gfSoundObject class permits a sound to be tethered to any object in the environment. Most sounds in a virtual environment will emanate from a physical source, graphically rendered to the observer in the environment. A gfSoundObject can be tethered to an

object and as the object is moved or updated in the environment, the sound will dynamically move with it.

Another significant option for gfSoundObjects is the ability to network sounds. For example, if two participants share a networked virtual world, and the first user initiates an action that produces a sound, the second user should hear the sound in the exact location where the first user created it. The gfSoundObject contains a state variable for use by the developer to automatically inform all remote users in a shared virtual world of a sound, as show in Figure 18.

```
void gfSoundObject::Send(gfSoundActionEnum action)
{
       gfSoundActionPacket *soundAction = new gfSoundActionPacket();
       strcpy(soundAction->mFileName, GetFileName());
       soundAction->action = action;
       sgVec3 pos;
       this->GetPosition(pos);
       sgSetVec3(soundAction->mPos, pos[0], pos[1], pos[2]);
       soundAction->mType = GFPACKET_TYPE_SOUND_ACTION;
       soundAction->mPacketSize = sizeof(gfSoundActionPacket);
       if(mTether) soundAction->mTethered = true;
       if(g_gfNetwork) g_gfNetwork->Send(soundAction);
}
```

Figure 18. gfSoundObject Send Method.

When the application developer desires to expose a sound to the network, the instantiation of the gfSoundObject constructor sets a variable that flags this sound as one that will transmit state information to the network whenever a state change is made. Figure 16 depicts the creation of a network data packet for the sound in question.

4. gfAudioEnvironment

The gfAudioEnvironment class represents an audio environment or room within the virtual world. A gfAudioEnvironment is instantiated for each separate room in the virtual environment, and individual acoustic characteristics can be set to represent the natural acoustics of that type of room or space. The gfAudioEnvironment.h and gfAudioEnvironment.cpp files contain two utility shape classes, gfCube (representing a three-dimensional cubic volume) and gfSphere (representing a three-Dimensional sphere) for determining the size and location of the audio environment.

The gfAudioEnvironment class is used strictly to determine whether a position in the virtual environment is contained within the associated shape of that environment and to store characteristics of that audio environment, such as size, shape, EAX™ acoustic properties and material conditions. The EAX™ acoustic properties are associated with the type of reverberation that will be applied in that audio environment. The material properties are settings that will affect the transitivity of the sound through barriers of the environment. For example, the material properties of an environment might be set to stone, indicating that the audio environment is mimicking a room with stonewalls, ceilings and floors. A different audio environment might be set to wood. A stone room will almost completely block all sounds generated inside the room from transmitting through the barriers walls. A wool room will permit a portion of the sound energy to transmit through the walls. Any position can be checked as to whether it resides inside the environment or outside the environment. For example, every time step the listener's position is compared to a global list of audio environments to determine which environment the listener is in. When a match is found, any characteristics of that environment can be set for the listener. The gfAudioEnvironment class, by itself, does not set the characteristics for any other class; it is used by the gfAudioEnvironmentManager to maintain the state of the listener as it moves through the virtual world.

5. gfAudioEnvironmentTransition

The gfAudioEnvironmentTransition class is used to model the acoustic effects of sounds as they transmit through portals, such as doorways and windows. It sets an area or location in the virtual environment where acoustic effects can be blended between audio environments with different characteristics.

Initially, a gfAudioEnvironmentTransition is constructed with references to two gfAudioEnvironments. The two environments are independently created to represent the separate audio environments separated by the portal. Additionally, like in gfAudioEnvironment, a shape is given to the transition zone overlapping the two environments, as in Figure 18. The same shape classes are used as in

gfAudioEnvironment. If the listener is located within a gfAudioEnvironmentTransition shape, the acoustic environment settings from each of the gfAudioEnvironments are blended together according to the relative positioning of the listener within the transition zone and the individual settings of the environments.

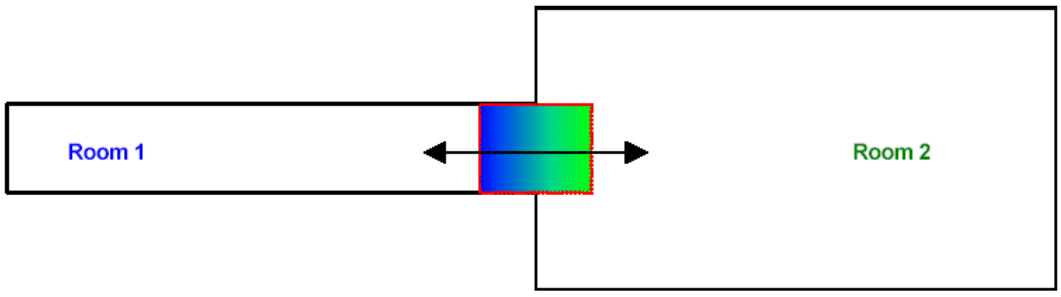


Figure 19. Audio Transition Zone.

Once it is determined that the listener is in a transition zone, the transitional effect must be created. Since EAX™ effects are generated through numeric inputs to EAX™ methods, performing a relative distance calculation on the numeric inputs for each of the two individual gfAudioEnvironments creates the blended effect. For example, if the listener is moving from one gfAudioEnvironment to another through a transition zone, and has moved through 25% of the transition zone then the resulting numeric inputs to EAX™ are calculated by simple mathematical averaging of 25% of the first gfAudioEnvironment's inputs with 75% of the second gfAudioEnvironment's inputs.

The calculations for the blended acoustic effect utilize the `EAXListenerInterpolate` function provided in EAX™ 3.0. Figure 19 shows the gfAudioEnvironmentTransition method for setting the interpolated effect. Both environments must be valid gfAudioEnvironments for transition effects to be applied. This precludes setting an effect when a transition zone is created without two corresponding gfAudioEnvironments.

```
void gfAudioEnvironmentTransition::SetTransitionEffect()
{
	if(!mEnv1 || !mEnv2) return;  //ensure both environments valid

	EAXLISTENERPROPERTIES transProps;  // for resulting properties
	EAXLISTENERPROPERTIES StartEAX3LP = GetEAXListenerProp(
		mEnv1->GetEnvironmentType());
```

```
        EAXLISTENERPROPERTIES FinishEAX3LP = GetEAXListenerProp(
               mEnv2->GetEnvironmentType());
        EAX3ListenerInterpolate(&StartEAX3LP, &FinishEAX3LP,
               GetTransitionRatio(), &transProps, true);
        g_gfListener->SetEnviron(transProps);
}
```

Figure 20. SetTransitionEffect Method.

First, the numeric parameters of each environment are extracted and provided to the `EAXListenerInterpolate` method with a ratio of relative distance moved through the zone. The amount of distance moved through the zone is obtained through the `GetTransitionRatio()` method, which uses gfAudioEnvironment environment shape center positions, gfAudioEnvironmentTransition shape center positions, and distances from the listener position to each of these points in determining what ratio of transition zone has been crossed and in which direction. Finally, the newly calculated numeric inputs are provided to the gfListener class to set a blended acoustic effect for the portal environment.

6. gfAudioEnvironmentManager

The gfAudioEnvironmentManager class is responsible for maintaining an accurate spatial relationship between the listener and all of the sounds in the environment. This class if instantiated constantly monitors the listener positions and compares it to the location of all sounds, whether statically positioned or dynamically updated. At a user prescribed rate, calculations of relative positions result in the setting of occlusion, obstruction, or exclusion effects for each individual sound. Additionally, the environment setting for the listener is automatically updated as the location of the listener moves through gfAudioEnvironments. The gfAudioEnvironmentManager class requires the presence of at least one gfAudioEnvironment for automatic processing of environmental effects, including audio environments and transition zones.

The primary effect of the gfAudioEnvironmentManager class resides in the Update() method, periodically calculates the actual positioning of the listener and sounds in the environment. Figure 20 depicts a small portion of the `Update()` method with pseudo-code in appropriate places. Occlusion, obstruction and exclusion are effects

applied to each sound depending on whether the position of those sounds is within eyesight of the listener and on whether the listener and the sound reside in the same or different gfAudioEnvironments. If the listener and the sound exist in the same audio environment, they can be considered to be in the same room. If they reside in different audio environments, they are considered to be in different rooms. The position associated with a sound does not require the presence of a visible object. First, the `SetListenerEnvironment()` method determines what reverberation setting must be applied to the listener's global environment. Then, each sound in the environment is screened against the following criteria:

- The sound is playing
- The sound is looping
- The sound is not positioned relative to the listener

```
SetListenerAudioEnvironment();
CheckSounds();
if(SoundObjectList->GetNum() > 0)
{
	for(int i = 0; i < SoundObjectList->GetNum(); i++)
	{
		mTempSound  = (gfSoundObject*)SoundObjectList->Get(i);
		if(!mTempSound->IsRelative() && (mTempSound->IsPlaying() ||
			mTempSound->IsLooping()) )
		{
			mTempSound->GetPosition(soundPos);
			mTempAudEnv = GetSoundObjectAudioEnvironment();
			bool hit = gfGetLOS(listenerPos, soundPos);

	// in same audio environment and not obstructed - no effects
			if( hit && (mListenerEnv == mTempAudEnv))
			{
				Remove effects
			}

	// excluded - hit but not in same environment
			else if( hit && (mListenerEnv != mTempAudEnv))
			{
				Remove obstruction and occlusion
				Set exclusion effect
			}
	// obstructed - not hit but in same environment
			else if( !hit && (mListenerEnv == mTempAudEnv))
			{
				Remove exclusion and occlusion
				Set obstruction effect
			}

	// occluded - not hit and not in same environment
			else if( !hit && (mListenerEnv != mTempAudEnv))
			{
				Remove exclusion and obstruction effect
```

```
                    Set occlusion effect
                }
            }
        }
```

Figure 21. Portion of gfAudioEnvironmentManager Update() Method.

The purpose for screening out sounds that are set as relative to the listener is that these sounds are generally placed in close proximity to the listener to simulate such sounds as footsteps or the sound of a weapon fired by the listener. Since these sounds are generally positioned closely to the listener, effects such as obstruction and occlusion do not apply - there are extremely few cases where the sound of the listener's footsteps will be in a different audio environment than the listener's ears. To reduce processing and CPU overhead, these sounds are eliminated from obstruction, occlusion and exclusion

Sound within eyesight of listener	Sound and listener in same audio environment	No effect	
Sound within eyesight of listener	Sound and listener in different audio environments	Exclusion	Although within eyesight, the sound is outside the listener's audio environment. Example: listener in a room and the sound is in the hallway outside the room, but the position of the sound is visible to the listener Result: Listener receives direct path sound energy but no reverberated or reflected sound energy
Sound not within eyesight of listener	Sound and listener in same audio environment	Obstruction	Although the sound and listener are in the same environment, the sound position can not be seen by the listener Example: Both listener and sound in a room, but an object is between them precluding the listener from seeing the sound's position Result: Listener does not receive any direct path sound energy, only reverberated or reflected sound waves
Sound not within eyesight of listener	Sound and listener in different audio environments	Occlusion	The listener cannot see the sound's position and they are in different audio environments. Example: Listener in one room and sound in another, without a portal in direct LOS between listener and sound positions Result: Listener receives neither direct path reflected sound energy; only muffled sound energy transmitted through separating medium is heard

Table 2. Criteria for Determining EAX™ Effect.

effects processing. Once the list of sounds is screened, each sound's position is used to determine:

- Whether the sound's position is within eyesight of the listener, and
- Whether the sound and listener are in the same or different audio environments.

Table 2 shows the criteria for which type of effect (obstruction, occlusion or exclusion) to set for each remaining sound.

7. gfNetVoice

The gfNetVoice class is responsible for adding live voice to a virtual environment in the case where multiple players in the shared virtual world are distributed, precluding the use of the Ausim3D GoldServe for live voice streaming. gfNetVoice depends on the presence of a DirectPlay™ network instance for its data transportation between peers. The DirectPlay™ network connection is created in a gfNetwork class, which is part of the core gfLib development package. Establishment of the DirectPlay™ network, in the gfNetwork class, was placed in the core library for two reasons:

- A network for data communications may be required in certain gfLib applications even without the need for voice communications. If the gfNetwork class was part of the gfAudio library, application developers would be required to link yet another library in with their software when a major portion of the gfAudio library functionality was not required.
- Developer-derived networking classes, which would add functionality to gfNetwork, would also require the presence of the gfAudio library when gfAudio functionality was not required

Even though its networking "under-layer" is present in a different library, gfNetVoice accomplishes live streaming voice by creating a peer-to-peer voice session over the underlying network connection. The underlying network connection may also be used for data and state information passing between participants. This model, separating the network connection from the voice connection, is also the standard model used in VoIP applications. For example, JVOIP, a spatialized VoIP framework library used to add live voice to virtual environments, uses an Real-Time Transport Protocol (RTP) network connection as the data transport layer below the VoIP connection used for the streaming voice.

The gfNetVoice voice session is created as a peer-to-peer voice session. Each session has a host, which can be a different node in the network than the gfNetwork host. Both DirectPlay™ networks and DirectVoice™ voice sessions can be configured in two ways, either peer-to-peer or client-server. In client server configurations, all data and voice transmissions pass through a central server application and host machine. For live voice, this adds a considerable amount of latency in the voice signal, and precludes utilization. Because the natural selection for the voice session is peer-to-peer, an equivalent mode was selected for the DirectPlay™ network.

The host in the voice session is responsible for creating an **IDirectVoicePlayServer** object and a session description, **DVSESSIONDESC**. The session description is used to set parameters of the voice session; the most important parameter is the compression type used for converting the voice signal into binary data fro transmission over the DirectPlay™ network. Figure 21 depicts the session host setup.

```
if( FAILED( hr = CoCreateInstance(CLSID_DirectPlayVoiceServer, NULL,
       CLSCTX_INPROC_SERVER, IID_IDirectPlayVoiceServer,
       (LPVOID*) &voiceServer ) ) )
. . .
if( FAILED( hr = voiceServer->Initialize(g_Peer, ServerMessageHandler,
(void*)this, 0, 0 ) ) )

ZeroMemory(&SessionDesc, sizeof(DVSESSIONDESC));
SessionDesc.dwSize = sizeof(DVSESSIONDESC);
SessionDesc.dwFlags = DVSESSION_NOHOSTMIGRATION;
SessionDesc.dwSessionType = DVSESSIONTYPE_PEER;
SessionDesc.dwBufferQuality = DVBUFFERQUALITY_DEFAULT;
SessionDesc.guidCT = DPVCTGUID_ADPCM;
SessionDesc.dwBufferAggressiveness = DVBUFFERAGGRESSIVENESS_DEFAULT;

if( FAILED( hr = voiceServer->StartSession(&SessionDesc, 0 ) ) )
. . .
```

Figure 22. gfNetVoice Host Setup.

Both the host and the clients must connect to the voice session. While Figure 21 depicts source code found only in the host, Figure 22 shows both host and client construction of an **IDirectPlayVoiceClient** object. Additionally, two DirectVoice™ structs, `DVSOUNDDEVICECONFIG` and `DVCLIENTCONFIG`, set parameters for the host or client machine's sound card for recording, playback, window focus, and threshold volume. Threshold volume is used to start a recording on either the host or client. To preclude flooding the network with voice packets when no voice is present but

background sounds are heard, the threshold level can be modified to require a certain volume at the microphone before recording, compression and transmission commences.

```
if( FAILED( hr = CoCreateInstance(CLSID_DirectPlayVoiceClient, NULL,
             CLSCTX_INPROC_SERVER,
             IID_IDirectPlayVoiceClient,
             (LPVOID*) &voiceClient ) ) )
. . .

if( FAILED( hr = voiceClient->Initialize(g_Peer, ClientMessageHandler,
(void*)this, 0, 0 ) ) )

. . .

gfWindow *win = (gfWindow*)WinList->Get(0);
ZeroMemory(&SoundDeviceConfig, sizeof(DVSOUNDDEVICECONFIG));
SoundDeviceConfig.dwSize = sizeof(DVSOUNDDEVICECONFIG);
SoundDeviceConfig.dwFlags = DVSOUNDCONFIG_AUTOSELECT;
SoundDeviceConfig.guidPlaybackDevice = DSDEVID_DefaultPlayback;
SoundDeviceConfig.lpdsPlaybackDevice = pDS;
SoundDeviceConfig.guidCaptureDevice = DSDEVID_DefaultCapture;
SoundDeviceConfig.lpdsCaptureDevice = NULL;
SoundDeviceConfig.hwndAppWindow = (HWND)win->GetGZWindow()->getHandle
SoundDeviceConfig.lpdsMainBuffer = NULL;
SoundDeviceConfig.dwMainBufferFlags = 0;
SoundDeviceConfig.dwMainBufferPriority = 0;


ZeroMemory(&ClientConfig, sizeof(DVCLIENTCONFIG));
ClientConfig.dwSize = sizeof(DVCLIENTCONFIG);
ClientConfig.dwFlags = DVCLIENTCONFIG_AUTOVOICEACTIVATED |
       DVCLIENTCONFIG_AUTORECORDVOLUME | DVCLIENTCONFIG_MUTEGLOBAL |
       DVCLIENTCONFIG_ECHOSUPPRESSION  ;
ClientConfig.lRecordVolume = DVRECORDVOLUME_LAST;
ClientConfig.lPlaybackVolume = DVPLAYBACKVOLUME_DEFAULT;
ClientConfig.dwThreshold = DVTHRESHOLD_UNUSED;
ClientConfig.dwBufferQuality = DVBUFFERQUALITY_DEFAULT;
ClientConfig.dwBufferAggressiveness = DVBUFFERAGGRESSIVENESS_DEFAULT;
ClientConfig.dwNotifyPeriod = 0;

if( FAILED( hr = voiceClient->Connect(&SoundDeviceConfig, &ClientConfig,
DVFLAGS_SYNC ) ) )
. . .
```

Figure 23. gfNetVoice Client Connection Source Code.

After all parameters are set, the client connects to the voice server and the voice session is commenced. DirectVoice™ supports a voice session with up to sixty-four participants. However, it would be near impossible to discern sixty-four voices simultaneously speaking in an environment. Additionally, sixty-four clients would generate such a significant amount of voice data packets that the network would be saturated to the point of being unusable. However, in many scenarios involving military

virtual environment training systems, where two to ten participants train together, gfNetVoice is fully capable of providing live voice to all virtual environment users.

The gfNetVoice class also contains functionality to provide for the following:

- Spatialized voice - gfNetVoice can spatialize each individual voice in three dimensions. Spatialization is accomplished in much the same way as in gfSoundObject, through creation of IDirectSound3Dbuffers for each voice client and positioning those buffers as desired.
- Occlusion, obstruction and exclusion - gfNetVoice permits setting of EAX™ effects for individual voices in the virtual environment
- Minimum and maximum voice distance - gfNetVoice allows for setting of minimum and maximum voice distances, which can be used to alter the distance a voice is heard in the virtual environment

The gfNetVoice class is indirectly tied to the gfListener class. The gfListener is responsible for sending voice position information over the network for its respective user in the virtual environment. Every time the gfListener position is updated, a new voice position packet is generated and transmitted to all users in the shared virtual world. In this manner, voices are automatically tethered to listeners (mouths tethered to ears).

C. AUSERVERLIB

1. auSystem

The auSystem class is designed as an overall timer and administrator between all other classes, responsible for sending internal messages between classes, internal timing for managing updates to the GoldServe, and permitting developers to adjust timing as to not overload the GoldServe with updates faster than the GoldServe system will allow before latency in positioning occurs.

2. auBase

The auBase class is used as a base class for all derived au classes. It is derived from Gizmo3D classes, which permits instantiations of auBase-derived classes to be considered as message senders or receivers for internal message passing. Several methods are included to permit developers to determine actual class type when auBase-derived classes are used as objects in internal messages.

3. auSource

The auSource class represents an Ausdim3D GoldServe source. Two classes are derived from auSource that are used by developers - auSound and auChannel - which correspond to the two types of sources that the GoldServe operates with: sources created from pre-recorded wav files (auSound) and sources relating to live input channels (auChannel). Any source can be positioned in three dimensions, and can have source-specific rolloff attenuation settings. Volume or gain can be adjusted for any source. Unlike DirectSound3D™, where volume can only be reduced from the original recording level, auSources permit volume amplification beyond the original recorded level. Rolloff is a source-specific parameter for auSources, and can be adjusted for each source independently. Position, volume and rolloff for auChannels and auSounds are set through methods in the auSource class.

Like in DirectSound3D™, sources can either be spatialized or non-spatialized. Non-spatialized sources for the GoldServe can be thought of as 2D sounds in DirectSound3D™. For non-spatialized sources, programmers have the ability to stereo pan the source to the left or the right of the listener. However, no spatialization processing is conducted and the source will appear to move automatically with the listener. An example of this is the footsteps, discussed as an example of 2D sounds in DirectSound3D™. Additionally, when a sound is not spatialized, no externalization will be possible. Externalization is the perception to the user that the sound source originates from a position external to the listener. When a sound is panned, although it can appear to emanate to the left or the right of the listener, it will not appear to emanate from a left or right position displaced form the user.

The auSource class handles source directivity for auChannels and auSounds. Directivity of a sound is achieved by attenuating the radiation of a source at various angular measurements around the source (see Figure 23.)

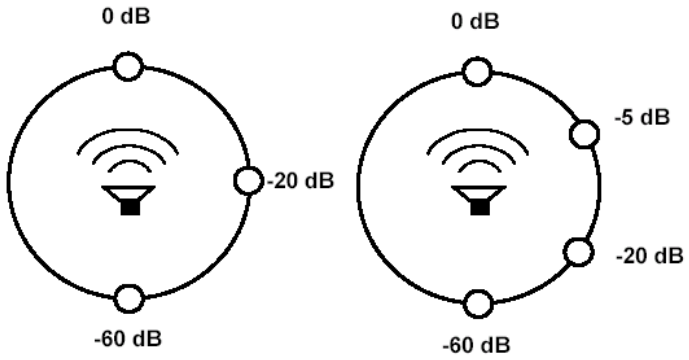


Figure 24. auSource Radiation Pattern Examples (www.ausim3d.com).

All other functionality of auChannels and auSounds is found in the respective individual classes.

4. auListener

The auListener class, like the gfListener, represents the ears of the observer in a virtual environment. Like in DirectSound3D™, the listener's position and orientation can be constantly updated in the virtual environment, providing an auditory experience of moving through the virtual world. The GoldServe Audio Localizing Server system used to develop the auServerLib software is capable of serving up to four concurrent listeners within an application, or single listeners supporting up to four simultaneously running separate applications. AuListeners are identified by name or by identification number. Listener identification numbers are created sequentially (starting with zero) as listeners are instantiated within an application.

The auListener class, unlike DirectSound3D™ and DirectVoice™, integrates live voice with the listener. A portion of the auListener `Config()` method is shown in Figure 24.

```
int mode = Atrn_METER | _VERBOSE_;
if(cre_init(Atrn_BMP1, mID|_ORATOR_ , 32,  mode) < Ok)
```

Figure 25. auListener Config() Method.

The mode variable sets the standard distance for rolloff calculations through the setting of the `Atrn_METER` flag. The `cre_init` method creates the listener (the Atrn_BMP1 flag is used to identify the type of operating software on the GoldServe; the GoldServe software is developed to operate on several different Ausim3D and legacy systems.) The `mID` variable represents the listener identification number. The `_ORATOR_` flag creates a live input source for the listener's voice that is continually re-positioned as the listener position is updated. Additionally, the listener is blocked from hearing input from the source identified as his or her own voice. All other listeners in the virtual environment will hear the voice signal. Additional functionality regarding the live input for the listener's voice includes:

- Voice volume. The global volume of any listener's voice input can be modified. If modified, the volume change will affect all other listener's perceptions of the volume.
- Voice radiation pattern. As with auSources, the radiation pattern of the live voice input can be modified for directivity (see Figure 23). This is useful in modeling the effect of a voice when the listener changes orientation. A live voice (tethered to a visual object, such as an avatar in the virtual environment) would sound different (generally louder) when the avatar is facing the listener than when the avatar is facing away from the listener (attenuated).
- Mouth offset. Our mouths and our ears are displaced from each other. If listeners were positioned in close proximity to one another, we would be able to discern the separation of the mouth (as a sound source) from our ears (as a reception source). To model this separation, mouth offset is used to place the live input voice source at a specified distance and orientation from the reception (ears) location.
- Voice input channel. This is a utility method for prescribing which of the external live input connections will service the microphone for the voice input of the listener.

In some circumstances, the volume of a listener's voice may not be of the same intensity for all other listeners in the virtual environment. For example, three users (listener A, listener B, listener C) navigate through a virtual environment, with two (listener A and listener B) being positioned inside the same room, and the third (listener C) positioned outside the room. Between listener A and listener B, voice volumes should be unaffected. However, the volume of the voice of listener A as heard by listener C should be attenuated (in addition to normal distance attenuation) due to occlusion, much

like in DirectSound3D™. Figure 25 displays a method in auListener that permits setting individual volumes for a listener's voice to any other listener in the virtual environment, mimicking the occlusion and obstruction effects found in EAX™. The IdAndGain struct permits the paring of a single listener identification number with a volume level. All other listeners in the virtual environment hear the voice volume as if it were unadjusted.

Additionally, virtual environment conditions may require that any single voice be directed only to a single exclusive listener. This can model radio communications where not all participants in a shared virtual environment hear the respective listener's voice; only the specified listener receives the voice audio, spatialized or non-spatialized. Figure 26 shows the auListener SetExclusive method for obtaining this functionality. Notice that setting a voice exclusive to a listener requires two operations; first, the voice volume is set to zero for all listeners in the virtual world (effectively muting the voice) and then the volume is adjusted to desired level for the specified listener.

```
bool auListener::SetVolumeForListener(auListener* otherListener, float dB)
{
	. . .
	IdAndGain temp;
	temp.id = otherListener->GetID();
	temp.dB = dB;

	if(cre_define_source(mID, AtrnPATHgain, sizeof(IdAndGain), &temp) < 0)
	{
		. . .
		return FAILURE;
	}
	. . .
	return SUCCESS;
}
```

Figure 26. auListener SetVolumeForListener Method.

```
bool auListener::SetExclusive(auListener* listener)
{
. . .
	IdAndGain temp;
	temp.id = ALL_HEADS;
	temp.dB = PATH_GAIN_DISABLE_PATH;

	if(cre_define_source(mID, AtrnPATHgain, sizeof(IdAndGain), &temp) < 0)
	{
		. . .
		return FAILURE;
	}

	temp.id = listener->GetID();
	temp.dB = 0.0f;

	if(cre_define_source(mID, AtrnPATHgain, sizeof(IdAndGain), &temp) < 0)
```

```
	{
		. . .
		return FAILURE;
	}
	return SUCCESS;
}
```

Figure 27. auListener SetExclusive Method.

5. auSound

An auSound instance represents a sound in the virtual environment in the form of a pre-recorded wav file. Sounds can be positioned in three dimensions, radiate in prescribed patterns, and attenuate with specific rolloff. Since these capabilities exist for both auSounds and auChannels, the code and functionality exists in the auSource class, the parent of both auSound and auChannel.

To hear a sound, it can either be played or looped. Looping a sound refers to a continuous playing of the wav file from start to finish over and over until a termination signal is sent. Figure 27 shows how to play or loop a sound. The only difference required to distinguish whether a sound is played or looped is which flag to set when calling the `cre_ctrl_wave` method; `WaveCTRL_STRT` indicates the sound should be played one time from start to completion, and `WaveCTRL_LOOP` indicates the sound should be continuously looped until termination.

```
bool auSound::Play()
{
	. . .
	if(cre_ctrl_wave(mSourceID, mWav, WaveCTRL_STRT, NULL) < 0)
	{
		. . .
		return FAILURE;
	}
	cre_update_audio();
	return SUCCESS;
}

bool auSound::Loop()
{
	. . .
	if(cre_ctrl_wave(mSourceID, mWav, WaveCTRL_LOOP, NULL) < 0)
	{
		. . .
		return FAILURE;
	}
	return SUCCESS;
}
```

Figure 28. auSound Play( ) and Loop( ) Methods.

In certain situations, it may be desirable to link a sound to a listener in the virtual environment. Since the listener represents the observer or participant in the environment, linking a sound to the listener essentially links a sound to the participant. This is useful in the case of footsteps, mentioned previously as a sound that will continuously move through the virtual environment with the observer. Figure 28 shows how to link a sound with a listener. The name of the listener is stored for later retrieval if necessary.

Like voices, sound volume can also be set individually for individual listeners, or can be set exclusive to a single listener in the virtual environment. To see an explanation of either function, see the auListener section or Appendix B.

```
bool auSound::LinkToListener(auListener* listener)
{
	mLinkListener = listener;
	isLinked = true;
	if(!isConfigured) return FAILURE;

	if(cre_define_source(mSourceID, AtrnHEADlink, listener->GetID(),
		NULL) < 0)
	{
		. . .
		isLinked = false;
		return FAILURE;
	}
	strcpy(mLinkName, listener->GetName());
	return SUCCESS;
}
```

Figure 29. auSound LinkToListener Method.

6. auChannel

The auChannel class represents a live input to the Ausim3D GoldServe. In this architecture, the live inputs represent the live voices of the participants in the shared virtual world. However, auChannels could be instantiated to add live audio inputs to a virtual environment such as radio transmissions, external streaming sounds, or any continuous audio signal.

Since much of the source code is very similar to that found in auListener or auSound, no source code is listed here. The auChannel class contains the necessary methods to permit:

- Linking live input channels with a listener. As with auSound, when an auChannel is linked to a specific auListener, the channel is automatically positioned with the auListener and updated accordingly.
- Setting the channel volume for specified listeners. Like the auListener `SetVolumeForListener()` method, individual volume settings may be made for each listener in the virtual environment.
- Setting the channel exclusive to a listener. A live input channel may be "routed" to only a single listener in the virtual environment.

The third capability mentioned, the ability to set a channel exclusive to a listener, is critical to the audio architecture prescribed in this thesis. In the scenario where multiple users connect to a networked, shared virtual environment and the participants are physically co-located, the architecture implements the Ausim3D GoldServe for live voice and uses the individual client PCs to generate the environment audio. To combine the individual audio from each user's machine with the live 3D voice supplied by the GoldServe, the individual user's audio, generated by the client PC's sound card using DirectSound3D™ and EAX™, is routed to the GoldServe as two live input channels, one for each stereo channel. Using the ability to set any channel exclusive to a listener, these live inputs are routed only to the respective listener in the virtual environment. Voice audio is layered over the live input, thereby providing a live voice capability to an audio environment. Since the Audim3D GoldServe is not currently capable of providing for environmental effects such as reverberation, occlusion and obstruction, using DirectSound3D™ and EAX™ on each client machine creates those necessary effects while the exceptionally low-latency live input capability of the GoldServe processes live voice. The architecture blends the most beneficial aspects of DirectSound3D™, EAX™ and hardware capability to provide the most immersive, interactive audio environment available.

7. auNotify

The auNotify class is a utility class used to print text and data, generally to a console window as part of an application. Developers can use one of five notification levels to determine the level of text output.

8. auTools

The auTools class contains methods to set global variables in the GoldServe audio environment, such as global rolloff factors and global atmospheric absorption rates affecting all sounds and live input channels. Additionally, the auTools class contains utility methods used by the auListener, auSound and auChannel classes to provide sequential source numbers used by the GoldServe for source and system management.

9. Summary

The auServerLib implementation's primary purpose is to permit the inclusion of live, low-latency, spatialized voice in a networked virtual environment. However, the software library's secondary purpose was to encapsulate the GoldServe's native programming API (CRE_TRON) into a useful programming suite of tools for use in other applications as well. The auServerLib can be used to create a complete, stand-alone auditory environment or can be used exclusively as the audio suite for a virtual environment. The main reason for not choosing the Ausim3D GoldServe for all audio in a virtual environment is its inability to provide effects, such as occlusion, obstruction, exclusion and reverberation, currently incorporated by EAX™. Ausim3D is currently developing a programming API that will incorporate all aspects of environmental acoustics.

VI. CONCLUSIONS AND RECOMMENDATIONS

A. SUMMARY

This thesis has provided guidance and an example of an audio architecture capable of serving various configurations of virtual environment training systems. From single, independent users to networked, multi-user shared virtual environments with live streaming voice, this architecture is capable of delivering a fully immersive, interactive, and high quality audio capability for inclusion in virtual environment training systems and simulators. The software component of this architecture is capable of operating on any standard PC and is developed from free or public-domain source code. The hardware component, the Ausim3D GoldServe, is COTS technology immediately available for utilization.

B. RECOMMENDATIONS

The architecture recommended by this thesis for use in virtual environment training systems is meant to support three virtual environment configurations:

- Single, independent user
- Multiple users, live voice not required
- Multiple users, physically co-located, live voice required
- Multiple users, physically distributed, live voice required

For single, independent users, a combination of DirectSound3D™ and EAX™ software operating on a single PC can provide the types of sounds and sound effects necessary to accurately simulate most acoustic environments found in military training scenarios. DirectSound3D™ encompasses the functionality to fully spatialize all sounds, providing dynamic distance attenuation, rolloff, volume and frequency manipulation, and source directivity. Since sound is interactive with the environment in which it is played, EAX™ provides the ability to model sound interaction with the physically modeled graphical environment to create effects such as reverberation, occlusion, obstruction, and exclusion. Without these effects, sounds in an environment will not appear to be realistic nor will they sound as they do in the real world. As seen in the two task analyses of military training evolutions, acoustic cues may be critical elements of a task. If a virtual

environment is to create a synthetic simulation of a training environment, the production of realistic audio cues, through the integration of DirectSound3D™ and EAX™, is a necessary element and component of the virtual environment training system.

When multiple users share a virtual environment where no live voice is required, the above configuration is still applicable. Because DirectSound3D™ is tightly integrated with DirectPlay™ within the DirectX development kit, multiple-user acoustic environments can be developed for up to sixty-four users.

For the two configurations mentioned thus far, where virtual environment audio is generated local to the respective user's machine and sound card, the quality and capability of the sound card is paramount. During development and testing of the implementation described in this thesis, Creative Technologies Audigy™ sound cards where utilized on all systems. The Audigy™ is the latest generation sound card released by Creative, and is widely considered as one of the best PC sound cards available today. Although DirectSound3D™ will work on just about every PC sound card commercially available, EAX™ requires a sound card supporting hardware acceleration through onboard hardware buffering. Additionally, since both EAX™ and the Audigy™ sound card are both produced by Creative, the Audigy™ is specially designed to support all EAX™ capabilities and provides the maximum number of hardware-accelerated buffers of most commercially available sound cards.

For multiple-user, networked virtual environments requiring live voice, there are two implementations of live voice that can be used. For shared virtual environment configurations where multiple users operate on individual machines but those machines are physically co-located, the Ausim3D GoldServe is by far the superior choice for live voice implementation. With considerably lower latency in voice processing than VoIP, the GoldServe offers the best solution for adding live voice to any multiple user virtual environments. Most deployed multiple user training systems will be confined to a small space or area for utilization, and the GoldServe equipment's size and footprint are extremely small. Integrating the superior sound and sound effects capabilities of DirectSound3D™ and EAX™ found in the single-user configuration with the low-latency

voice quality of the GoldServe offers the best audio configuration when users are physically co-located.

In those cases where multiple users must be widely distributed, VoIP provides the live voice component. Although VoIP faces critical voice latency issues, when users are distributed over a wide area, there are no other effective live voice implementations available. For deployed training systems, where even simple telephone implementations of live voice are architecturally impossible, VoIP is the only solution. DirectVoice™ is the only known VoIP application that fully supports spatialized live voice, and is the implementation recommended by this thesis.

C. FUTURE WORK

Although this study produced an audio architecture and implementation capable of providing a high quality auditory environment for virtual training systems, there are two areas of further research that will vastly improve the architecture's capability to provide an even better acoustic environment and live voice capability for distributed virtual environments.

First, while EAX™ is an extremely capable API for creating sound effects such as reverberation, occlusion, and obstruction, it is still somewhat limited in that it does not do actual acoustic modeling of the environment. As a parameter-based sound effects API, developers are required to manipulate variables to achieve desired sound effects, many times through trial and error, a pain-staking and time-consuming process. With five high-level parameters and over 15 low-level parameters, the number of permutations of variable changes and modifications in EAX™ possible to achieve a specific effect can be overwhelming. Considering that these calculations and manipulations must occur not just once for a virtual environment, but for every position a user could find himself or herself in the virtual environment, a realistic auditory environment in a complex virtual world may take months to program and develop. For true acoustic modeling, a geometry-based approach is necessary. Geometry-based acoustic modeling refers to modeling the acoustics and audio characteristics of an environment based on the actual virtual geometry of that environment, much akin to graphical ray tracing in the visual sense.

Sounds played in the environment will interact with the visual geometry to naturally, and automatically, produce reverberation, occlusion and obstruction effects. Scenario developers for virtual environment training systems will avoid the time-consuming process of manipulating numerous variables to achieve realistic sound effects. Instead, an algorithm would produce the effects based on information provided by the developer as to the characteristics of the environment. This information would only have to be provided once, as compared to the multiple iterations of calculations required in parameter-based sound effects. Research into developing a geometry-based acoustic modeling capability will not only vastly improve the quality of the effects, but also significantly reduce the production time for high quality auditory environments for virtual training systems.

Second, alternatives for live, low latency streaming voice must be developed. While the Ausim3D GoldServe provides an exceptionally low latency streaming voice capability, the requirement to be physically co-located with the hardware and the other participants in the virtual environment preclude its universal implementation. While many training systems, and most of the deployed systems, will in fact not suffer due to this limitation, there are many other virtual training systems under development designed to provide multiple-user or team training when participants or team members are distributed over large areas. One of the great benefits of virtual environments is their ability to place multiple individuals in a shared world even if they are not in a shared location to conduct simulations or training. Until a low latency solution for live voice is discovered, these distributed virtual environments will either be faced with suffering from a high latency live voice system or nor live voice at all. Neither are optimum, and further research and development into an IP-based, low latency live voice capability is necessary.

APPENDIX A. GFAUDIO DOCUMENTATION

A. GFAUDIOENVIRONMENT CLASS REFERENCE

1. Public Types

- **enum gfEnvironEnum { GF_GENERIC = 0, GF_PADDEDCELL, GF_ROOM, GF_BATHROOM, GF_LIVINGROOM, GF_STONEROOM, GF_AUDITORIUM, GF_CONCERTHALL, GF_CAVE, GF_ARENA, GF_HANGAR, GF_CARPETEDHALLWAY, GF_HALLWAY, GF_STONECORRIDOR, GF_ALLEY, GF_FOREST, GF_CITY, GF_MOUNTAINS, GF_QUARRY, GF_PLAIN, GF_PARKINGLOT, GF_SEWERPIPE, GF_UNDERWATER, GF_DRUGGED, GF_DIZZY, GF_PSYCHOTIC }**

  gfEnvironEnum - enum representing EAX™ effects.

- **enum gfMaterialEnum { GF_NONE = 0, GF_WINDOW, GF_DOOR, GF_WOOD, GF_BRICK, GF_STONE }**

2. Public Methods

- gfAudioEnvironment **(**gfEnvironEnum **environEnum=GF_GENERIC,** gfShape ***shape=NULL, const char *name=0)**

Constructor.

- **virtual ~gfAudioEnvironment ()**

Destructor.

- **virtual void SetEnvironmentType (gfEnvironEnum environEnum)**

SetEnvironmentType - sets the environment type.

- **gfEnvironEnum GetEnvironmentType ()**

GetEnvironmentType - gets the environment enum.

- **virtual void SetEnvironmentShape (gfShape *shape)**

SetEnvironmentShape - sets environment shape.

- **gfShape * GetEnvironmentShape ()**

GetEnvironmentShape - returns pointer to area shape object.

- **bool** InsideEnvironment **(sgVec3 pos)**

InsideEnvironment - indicates whether position is inside environment or not.

- **void** Location **(sgVec3 pos)**

Location - moves the environment shape.

- **float *** GetLocation **()**

GetLocation - returns the center location of the shape.

- **void SetEnvironmentMaterial (gfMaterialEnum material)**

SetEnvironmentMaterial - sets material settings for the environment.

- **gfMaterialEnum GetEnvironmentMaterial ()**

GetEnvironmentMaterial.

3. Public Attributes

- **GZ_DECLARE_TYPE_INTERFACE**

4. Detailed Description

Class: gfAudioEnvironment
Function: class to store coordinates, type, and size of one audio environment

5. Member Enumeration Documentation

- **enum gfAudioEnvironment::gfEnvironEnum**

gfEnvironEnum - enum representing EAX effects.

Enumeration values:

GF_GENERIC
GF_PADDEDCELL
GF_ROOM
GF_BATHROOM
GF_LIVINGROOM
GF_STONEROOM
GF_AUDITORIUM
GF_CONCERTHALL
GF_CAVE
GF_ARENA
GF_HANGAR
GF_CARPETEDHALLWAY
GF_HALLWAY
GF_STONECORRIDOR
GF_ALLEY
GF_FOREST
GF_CITY
GF_MOUNTAINS
GF_QUARRY
GF_PLAIN
GF_PARKINGLOT
GF_SEWERPIPE
GF_UNDERWATER
GF_DRUGGED
GF_DIZZY
GF_PSYCHOTIC

- **enum gfAudioEnvironment::gfMaterialEnum**

Enumeration values:

GF_NONE
GF_WINDOW
GF_DOOR
GF_WOOD
GF_BRICK
GF_STONE

6. Constructor and Destructor Documentation

- **gfAudioEnvironment::gfAudioEnvironment (gfEnvironEnum *environEnum* = GF_GENERIC, gfShape * *shape* = NULL, const char * *name* = 0)**
    - Constructor.
    - Function: Constructor
    - Purpose: creates new gfAudioEnvironment and adds it to global list
    - **Parameters:**
        - ***name*** - name for environment
        - ***environEnum*** - **gfAudioEnvironment** enumeration for environment type

- **gfAudioEnvironment::~gfAudioEnvironment () `[virtual]`**
    - Destructor.
    - Function: Destructor
    - Purpose: Destroys this gfAudioEnvironment

7. Member Function Documentation

- **gfMaterialEnum gfAudioEnvironment::GetEnvironmentMaterial () `[inline]`**
    - GetEnvironmentMaterial.

- **gfShape* gfAudioEnvironment::GetEnvironmentShape () `[inline]`**
    - GetEnvironmentArea - returns box with area coordinates.

- **gfEnvironEnum gfAudioEnvironment::GetEnvironmentType () `[inline]`**
    - GetEnvironmentType - gets the environment enum.

- **float* gfAudioEnvironment::GetLocation () `[inline]`**
    - GetLocation - returns the center location of the shape.

- **bool gfAudioEnvironment::InsideEnvironment (sgVec3 *pos*)**
    - InsideEnvironment.
    - Function: InsideEnvironment
    - Purpose: indicates whether gfPosition lies inside shape of environment or not
    - **Parameters:**

        - ***pos*** - the gfPosition to check inside/outside
    - **Returns:**
        - bool - TRUE = inside; FALSE = outside

- **void gfAudioEnvironment::Location (sgVec3 *pos*)**
    - Locate - moves the environment shape.
    - Function: Locate
    - Purpose: locate an audio environment at a specified location
    - **Parameters:**
        - ***pos*** - the sgVec3 for the center position

- **void gfAudioEnvironment::SetEnvironmentMaterial (gfMaterialEnum *material*)**
    - SetEnvironmentMaterial.
    - Function: SetEnvironmentMaterial
    - Purpose: sets the environment material for occlusion
    - **Parameters:**
        - ***material*** - the enumeration for the specified environment material settings

- **void gfAudioEnvironment::SetEnvironmentShape (gfShape * *shape*) `[virtual]`**
    - SetEnvironmentArea - sets environment sphere.
    - Function: SetEnvironmentShape
    - Purpose: sets the environment shape
    - **Parameters:**
        - ***shape*** - the shape object defining the shape of the environment

- **void gfAudioEnvironment::SetEnvironmentType (gfEnvironEnum *environEnum*) `[virtual]`**
    - SetEnvironment - sets the environment type.
    - Function: SetEnvironmentType
    - Purpose: sets the environment type
    - **Parameters:**
        - ***environEnum*** - the enumeration for the specified environment

8. Member Data Documentation

gfAudioEnvironment::GZ_DECLARE_TYPE_INTERFACE

The documentation for this class was generated from the following files:

gfaudioenvironment.h
gfaudioenvironment.cpp

B. GFAUDIOENVIRONMENTMANAGER CLASS REFERENCE

```
#include <gfaudioenvironmentmanager.h>
```

1. Public Methods

- **gfAudioEnvironmentManager ()**

Constructor.

- **~gfAudioEnvironmentManager ()**

Destructor.

- **const char *** GetListenerEnv **()**

GetListenerEnv - gets a handle to the current listener environment.

2. Public Attributes

- **GZ_DECLARE_TYPE_INTERFACE**

3. Detailed Description

Class: gfAudioEnvironmentManager

Function: Class to manage setting environmental effects for the listener depending on listener location within gfAudioEnvironments or gfAudioEnvironmentTransitions. Manages automatic setting of obstruction, occlusion or exclusion settings for individual sounds and voices.

4. Constructor and Destructor Documentation

- **gfAudioEnvironmentManager::gfAudioEnvironmentManager ()**
    - Constructor.
    - Function: Constructor
    - Purpose: creates new **gfAudioEnvironment**

- **gfAudioEnvironmentManager::~gfAudioEnvironmentManager ()**
    - Destructor.
    - Function: Destructor
    - Purpose: destroys gfAudioEnvironmentManager

5. Member Function Documentation

- **const char* gfAudioEnvironmentManager::GetListenerEnv () `[inline]`**
    - GetListenerEnv - gets a handle to the current listener environment.

6. Member Data Documentation

- **gfAudioEnvironmentManager::GZ_DECLARE_TYPE_INTERFACE**

The documentation for this class was generated from the following files:

- **gfaudioenvironmentmanager.h**
- **gfaudioenvironmentmanager.cpp**

C. GFAUDIOENVIRONMENTTRANSITION CLASS REFERENCE

`#include <gfaudioenvironmenttransition.h>`

1. Public Methods

- **gfAudioEnvironmentTransition (gfAudioEnvironment *env1=NULL, gfAudioEnvironment *env2=NULL, gfShape *shape=NULL, const char *name=NULL)**

Constructor.

- **virtual ~gfAudioEnvironmentTransition ()**

Destructor.

- **void SetAudioEnvironmentOne (gfAudioEnvironment *env)**

SetAudioEnvironmentOne - sets first audio environment.

- **void SetAudioEnvironmentTwo (gfAudioEnvironment *env)**

SetAudioEnvironmentTwo - sets second audio environment.

- **gfAudioEnvironment * GetAudioEnvironmentOne ()**

GetAudioEnvironmentOne - gets first audio environment.

- **gfAudioEnvironment * GetAudioEnvironmentTwo ()**

GetAudioEnvironmentTwo - gets second audio environment.

- **void SetAudioEnvironmentTransitionShape (gfShape *shape)**

SetAudioEnvironmentTransitionShape - sets the shape.

- **gfShape * GetAudioEnvironmentTransitionShape ()**

GetAudioEnvironmentTransitionShape - gets the shape.

- **bool InsideEnvironmentTransition (sgVec3 pos)**

InsideEnvironmentTransition - checks whether pos is inside transition zone.

- **void SetTransitionEffect ()**

SetTransitionEffect - combines and morphs two EAX environments.

2. Public Attributes

- **GZ_DECLARE_TYPE_INTERFACE**

3. Detailed Description

Class: gfAudioEnvironmentTransition

Function: class to store coordinates and type of two audio environments for morphing between two audio reverb effects as listener moves between environments.

4. Constructor and Destructor Documentation

- **gfAudioEnvironmentTransition::gfAudioEnvironmentTransition (gfAudioEnvironment * *env1* = NULL, gfAudioEnvironment * *env2* = NULL, gfShape * *shape* = NULL, const char * *name* = NULL)**
    - Constructor.
    - Function: Constructor
    - Purpose: creates new gfAudioEnvironmentTransition

- **gfAudioEnvironmentTransition::~gfAudioEnvironmentTransition() `[virtual]`**
    - Destructor.
    - Function: Destructor
    - Purpose: destroys gfAudioEnvironmentTransition

5. Member Function Documentation

- **gfAudioEnvironment* gfAudioEnvironmentTransition::GetAudioEnvironmentOne () `[inline]`**
    - GetAudioEnvironmentOne - gets first audio environment.

- **gfShape* gfAudioEnvironmentTransition::GetAudioEnvironmentTransitionShape () `[inline]`**
    - GetAudioEnvironmentTransitionShape - gets the shape.

- **gfAudioEnvironment* gfAudioEnvironmentTransition::GetAudioEnvironmentTwo () `[inline]`**
    - GetAudioEnvironmentTwo - gets second audio environment.

- **bool gfAudioEnvironmentTransition::InsideEnvironmentTransition (sgVec3 *pos*)**
    - InsideEnvironmentTransition - checks whether pos is inside transition zone.
    - Function: InsideEnvironmentTransition
    - Purpose: indicates whether gfPosition lies inside shape of environment transition or not
    - **Parameters:**
        - ***pos*** - the sgVec to check inside/outside
    - **Returns:**
        - bool - TRUE = inside; FALSE = outside

- **void gfAudioEnvironmentTransition::SetAudioEnvironmentOne (gfAudioEnvironment * *env*)**
    - SetAudioEnvironmentOne - sets first audio environment.
    - Function: SetAudioEnvironmentOne
    - Purpose: sets audio environment one

- **Parameters:**
  - ***env*** **- gfAudioEnvironment** one

- **void gfAudioEnvironmentTransition:: SetAudioEnvironmentTransitionShape (gfShape * *shape*)**
  - SetAudioEnvironmentTransitionShape - sets the shape.
  - Function: SetAudioEnvironmentTransitionShape
  - Purpose: sets audio environment transition zone shape
  - **Parameters:**
    - ***shape*** - gfAudioEnvironmentTransition shape

- **void gfAudioEnvironmentTransition::SetAudioEnvironmentTwo (gfAudioEnvironment * *env*)**
  - SetAudioEnvironmentTwo - sets second audio environment.
  - Function: SetAudioEnvironmentTwo
  - Purpose: sets audio environment two
  - **Parameters:**
    - ***env*** **- gfAudioEnvironment** two

- **void gfAudioEnvironmentTransition::SetTransitionEffect ()**
  - SetTransitionEffect - combines and morphs two EAX environments.
  - Function: SetTransitionEffect
  - Purpose: sets the transitional reverb effect if in a transition zone

6. Member Data Documentation

- **gfAudioEnvironmentTransition::GZ_DECLARE_TYPE_INTERFACE**

The documentation for this class was generated from the following files:

- **gfaudioenvironmenttransition.h**
- **gfaudioenvironmenttransition.cpp**

D. GFAUDIONET CLASS REFERENCE

```
#include <gfaudioglobal.h>
```

1. Public Methods

- **gfAudioNet ()**
- **virtual** ~gfAudioNet **()**

2. Public Attributes

- **GZ_DECLARE_TYPE_INTERFACE**

3. Constructor and Destructor Documentation

- **gfAudioNet::gfAudioNet ()**
- **gfAudioNet::~gfAudioNet () `[virtual]`**

4. Member Data Documentation

- **gfAudioNet::GZ_DECLARE_TYPE_INTERFACE**

The documentation for this class was generated from the following files:

- **gfaudioglobal.h**
- **gfaudioglobal.cpp**

E. GFCUBE CLASS REFERENCE

`#include <gfaudioenvironment.h>`

1. Public Methods

- gfCube **(float minX=0.0f, float maxX=1.0f, float minY=0.0f, float maxY=1.0f, float minZ=0.0f, float maxZ=1.0f)**

Constructor.

- **~gfCube ()**

Destructor.

- **void** SetCube **(float minX, float maxX, float minY, float maxY, float minZ, float maxZ)**

SetCube - sets the coordinates for the environment cube.

- **virtual void** SetLocation **(sgVec3 pos)**

SetLocation - sets the location for the shape.

- **bool** Contains **(sgVec3 pos)**

Contains - determines whether position inside cube.

- **void** GetCenter **(sgVec3 pos)**

GetCenter - gets the center point of the cube.

- **void** Print **()**

2. Public Attributes

- **GZ_DECLARE_TYPE_INTERFACE**

3. Detailed Description

Class: gfCube Function: class for cube audio shape - aligned on coordinate axes

4. Constructor and Destructor Documentation

- **gfCube::gfCube (float *minX* = 0.0f, float *maxX* = 1.0f, float *minY* = 0.0f, float *maxY* = 1.0f, float *minZ* = 0.0f, float *maxZ* = 1.0f)**
    - Constructor.
    - Function: Constructor
    - Purpose: creates new gfCube
    - **Parameters:**
        - ***minX*** - environment cube minimum x coordinate
        - ***maxX*** - environment cube maximum x coordinate
        - ***minY*** - environment cube minimum y coordinate
        - ***maxY*** - environment cube maximum y coordinate
        - ***minZ*** - environment cube minimum z coordinate
        - ***maxZ*** - environment cube minimum z coordinate

- **gfCube::~gfCube ()**
    - Destructor.
    - Function: Destructor
    - Purpose: destroys gfCube

5. Member Function Documentation

- **bool gfCube::Contains (sgVec3 *pos*) `[virtual]`**
    - Contains - determines whether position inside cube.
    - Function: Contains
    - Purpose: determine whether the gfPosition is contained in the shape
    - **Parameters:**
        - ***pos*** - the position to check for containment
    - **Returns:**
        - bool - TRUE = inside shape; FALSE = outside shape
    - Implements **gfShape**.

- **void gfCube::GetCenter (sgVec3 *pos*) `[virtual]`**
    - GetCenter - gets the center point of the cube.
    - Function: GetCenter
    - Purpose: determine the center point of the cube
    - **Parameters:**
        - ***pos*** - the position to fill in data
    - Implements **gfShape**

- **void gfCube::Print ()**
    - Function: Print
    - Purpose: print the cubes coordinates

- **void gfCube::SetCube (float *minX*, float *maxX*, float *minY*, float *maxY*, float *minZ*, float *maxZ*)**
    - SetBox - sets the coordinates for the environment box.
    - Function: SetCube
    - Purpose: Creates a set of cube coordinates
    - **Parameters:**
        - ***minX*** - environment cube minimum x coordinate
        - ***maxX*** - environment cube maximum x coordinate
        - ***minY*** - environment cube minimum y coordinate
        - ***maxY*** - environment cube maximum y coordinate
        - ***minZ*** - environment cube minimum z coordinate
        - ***maxZ*** - environment cube minimum z coordinate

- **void gfCube::SetLocation (sgVec3 *pos*) `[virtual]`**
    - SetLocation - sets the location for the shape.
    - Function: SetLocation
    - Purpose: sets the center location for the cube
    - **Parameters:**
        - ***pos*** - the position of the center of the cube
    - Implements **gfShape**

6. Member Data Documentation

- **gfCube::GZ_DECLARE_TYPE_INTERFACE**

Reimplemented from **gfShape**

The documentation for this class was generated from the following files:

- **gfaudioenvironment.h**
- **gfaudioenvironment.cpp**

F. GFLISTENER CLASS REFERENCE

`#include <gfListener.h>`

1. Public Methods

- gfListener **(const char *name=0)**

Constructor.

- **~gfListener ()**

Destructor.

- **void** Position **(gfPosition *pos)**

Position - positions the listener.

- **void** SetObserver **(gfObserver *obs)**

SetObserver - sets gfObserver to tether with.

- **void** SetObserver **(const char *name)**

SetObserver - sets gfObserver to tether with (by name).

- **void** SetDopplerFactor **(float value)**

SetDopplerFactor - exaggerates doppler effect.

- **void** SetRolloff **(float value)**

SetRolloff - sets global rolloff.

- **void** SetVelocity **(sgVec3 vel)**

SetVelocity - velocity used in doppler effect.

- **void** SetVelocity **(sgVec3 dir, float spd)**

SetVelocity - velocity used in doppler effect.

- **void SetEnviron (gfAudioEnvironment::gfEnvironEnum env)**

SetEnviron - places listener in EAX environment using gfEnum.

- **void** SetEnviron **(EAXLISTENERPROPERTIES props)**

SetEnviron - places the listener in EAX environment using EAX's EAXLISTENERPROPERTIES struct.

- **void** SetEnvironSize **(float size)**

SetEnvironSize - sets the size of the room.

- **void** Shutdown **()**

Shutdown - shuts down the listener.

- **void CommitDeferredSettings ()**

CommitDeferredSettings - commits all deferred settings.

2. Public Attributes

- **GZ_DECLARE_TYPE_INTERFACE**

3. Protected Methods

- **void** Config **(void)**

Config - configures listener.

- **virtual gzVoid** onNotify **(gzNotifyMessage *message)**

onNotify - internal messaging.

4. Protected Attributes

- **LPDIRECTSOUNDBUFFER** pDSB
- **LPDIRECTSOUND3DLISTENER** pDSListener
- **LPDIRECTSOUND3DBUFFER** pDSB3D
- **LPKSPROPERTYSET** pEAXListener
- **gzRefPointer< gfObserver >** mObserver
- **sgVec3 mVelocity**
- **float** mVolume
- **bool isConfigured**
- **float mRolloff**
- **float mDoppler**
- **bool eaxSupported**

5. Detailed Description

Class: gfListener

Function: class to provide for a movable, dynamic listener with 3 dimensions. Contains methods to get/set positions, orientations, Doppler factor, velocity, and rolloff. Contains functionality to tether to a gfObserver for automatic positioning.

- **only one listener per context**

6. Constructor and Destructor Documentation

- **gfListener::gfListener (const char * *name* = 0)**
    - Constructor.
    - Function: Constructor
    - Purpose: creates new gfListener
    - **Parameters:**
        - ***name*** - optional name for listener

- **gfListener::~gfListener ()**
    - Destructor.
    - Function: Destructor
    - Purpose: Destroy this listener

7. Member Function Documentation

- **void gfListener::CommitDeferredSettings ()**
    - CommitDeferredSettings - commits all deferred settings.
    - Function: SetEnviron
    - Purpose: Sets the EAX reverb model
    - **Parameters:**
        - ***env*** - gfEnvironEnum representing selected EAX reverb model

- **void gfListener::Config (void)** `[protected]`
    - Config - configures listener.
    - Function: Config

- Purpose: configures DirectSound objects and interfaces

- **gzVoid gfListener::onNotify (gzNotifyMessage * *message*) `[protected, virtual]`**
  - onNotify - internal messaging.
  - Function: onNotify
  - Purpose: method called by gfObserver when position is updated
  - **Parameters:**
    - ***message*** message from gfObserver notifying listener of position update

- **void gfListener::Position (gfPosition * *pos*)**
  - Position - positions the listener.
  - Function: Position
  - Purpose: Sets the position and orientation of the listener
  - **Parameters:**
    - ***pos*** - gfPosition with both position and orientation information

- **void gfListener::SetDopplerFactor (float *factor*)**
  - SetDopplerFactor - exaggerates doppler effect.
  - Function: SetDopplerFactor
  - Purpose: Sets the Doppler factor for the listener - factor exaggerates real-world doppler effect
  - **Parameters:**
    - ***value*** - the multiplicative factor to apply to real-world Doppler calculations Default: 1.0

- **void gfListener::SetEnviron (EAXLISTENERPROPERTIES *props*)**
  - SetEnviron - places the listener in EAX environment using EAX's EAXLISTENERPROPERTIES struct.
  - Function: SetEnviron
  - Purpose: Sets the EAX reverb model
  - **Parameters:**
    - ***props*** - EAXLISTENERPROPERTIES representing selected EAX reverb model

- **void gfListener::SetEnviron (gfAudioEnvironment::gfEnvironEnum *env*)**
  - SetEnviron - places listener in EAX environment using gfEnum.
  - Function: SetEnviron
  - Purpose: Sets the EAX reverb model
  - **Parameters:**
    - ***env*** - gfEnvironEnum representing selected EAX reverb model

- **void gfListener::SetEnvironSize (float *size*)**
  - SetEnvironSize - sets the size of the room.
  - Function: SetEnvironSize

- Purpose: Sets the apparent room size only for EAX processing
- **Parameters:**
  - ***size*** - the virtual room size for EAX

- **void gfListener::SetObserver (const char * *name*)**
  - SetObserver - sets gfObserver to tether with (by name).
  - Function: SetObserver
  - Purpose: Tethers this listener to a gfObserver; listener maintains positions with observer
  - **Parameters:**
    - ***name*** - name of the observer to link with

- **void gfListener::SetObserver (gfObserver * *obs*)**
  - SetObserver - sets gfObserver to tether with.
  - Function: SetObserver
  - Purpose: Tethers this listener to a gfObserver; listener maintains positions with observer
  - **Parameters:**
    - ***obs*** - reference to the gfObserver to link with

- **void gfListener::SetRolloff (float *factor*)**
  - SetRolloff - sets global rolloff.
  - Function: SetRolloff
  - Purpose: sets the sound rolloff factor. The rolloff factor has a range of DS3D_MINROLLOFFFACTOR (0-no rolloff) to DS3D_MAXROLLOFFFACTOR (as currently defined, 10 times the rolloff found in the real world). The default value is DS3D_DEFAULTROLLOFFFACTOR (1.0).
  - **Parameters:**
    - ***factor*** - from DS3D_MINROLLOFFFACTOR(0) to DS3D_MAXROLLOFFFACTOR(10)

- **void gfListener::SetVelocity (sgVec3 *dir*, float *spd*)**
  - SetVelocity - velocity used in doppler effect.
  - Function: SetVelocity
  - Purpose: Sets the velocity for the listener - used in Doppler calculations
  - **Parameters:**
    - ***dir*** - direction vector for velocity
    - ***spd*** - speed for velocity calculations

- **void gfListener::SetVelocity (sgVec3 *vel*)**
  - SetVelocity - velocity used in doppler effect.
  - Function: SetVelocity
  - Purpose: Sets the velocity for the listener - used in Doppler calculations
  - **Parameters:**
    - ***vel*** - velocity vector

- **void gfListener::Shutdown ()**
    - Shutdown - shuts down the listener.
    - Function: Shutdown
    - Purpose: shuts down listener and its DirectSound objects

8. Member Data Documentation

- **bool gfListener::eaxSupported [protected]**
    - indicates whether EAX is supported
- **gfListener::GZ_DECLARE_TYPE_INTERFACE**
- **bool gfListener::isConfigured [protected]**
    - indicates whether listener is configured
- **float gfListener::mDoppler [protected]**
    - doppler factor
- **gzRefPointer<gfObserver> gfListener::mObserver [protected]**
    - observer this listener is tethered to
- **float gfListener::mRolloff [protected]**
    - rolloff factor
- **sgVec3 gfListener::mVelocity [protected]**
    - current velocity of listener
- **float gfListener::mVolume [protected]**
    - volume of listener - global setting affecting all sources
- **LPDIRECTSOUNDBUFFER gfListener::pDSB [protected]**
- **LPDIRECTSOUND3DBUFFER gfListener::pDSB3D [protected]**
    - DirectSound3D buffer - used to obtain EAX property set interface
- **LPDIRECTSOUND3DLISTENER gfListener::pDSListener [protected]**
    - DirectSound listener
- **LPKSPROPERTYSET gfListener::pEAXListener [protected]**
    - EAX property set interface

The documentation for this class was generated from the following files:

- **gfListener.h**
- **gfListener.cpp**

G. GFNETVOICE CLASS REFERENCE

`#include <gfnetvoice.h>`

1. Public Methods

- gfNetVoice **(bool host, const char *name=0)**

Constructor.

- **virtual** ~gfNetVoice **()**

Destructor.

- **void** SetMinVoiceDistance **(float value)**

SetMinVoiceDistance - Sets the distance at which no further gain is applied moving towards the voice object.

- **void** SetMaxVoiceDistance **(float value)**

SetMaxVoiceDistance - Sets the maximum voice distance - distance at which no further attenuation occurs.

- **void** SetOcclusion **(int voice, long occlusion, float occlusionLF, float occlusionRoomRatio)**

SetOcclusion - sets the specified voice to be occluded.

- **void** RemOcclusion **(int voice)**

RemOcclusion - removes occlusion from the specified voice.

- **bool** IsOccluded **(int voice)**

IsOccluded - indicates whether voice is occluded.

- **void** SetObstruction **(int voice, long obstruction, float obstructionLF)**

SetObstruction - sets obstruction values for this sound.

- **void** RemObstruction **(int voice)**

RemObstruction - removes obstruction from this sound.

- **bool** IsObstructed **(int voice)**

IsObstructed - indicates whether voice is obstructed.

- **void** SetExclusion **(int voice, long exclusion, float exclusionLF)**

SetExclusion - sets exclusion values for this sound.

- **void** RemExclusion **(int voice)**

RemExclusion - removes exclusion from this sound.

- **bool** IsExcluded **(int voice)**

IsExcluded - indicates whether voice is excluded.

- **int GetNumVoices ()**

GetNumVoices.

- **void** GetVoicePosition **(sgVec3 xyz, int Idx)**

GetVoicePosition - gets the position of the voice.

2. Detailed Description

Class: gfNetVoice

Function: class to provide for a movable, dynamic live voice with up to 63 remote voice clients. Contains methods to set minimum and maximum voice distances. Manages setting occlusion, obstruction or exclusion for individual voices.

3. Constructor and Destructor Documentation

- **gfNetVoice::gfNetVoice (bool *host*, const char * *name* = 0)**
    - Constructor.
    - Function: Constructor
    - Purpose: creates new gfNetVoice
    - **Parameters:**
        - ***host*** - bool indicating whether this gfNetVoice hosts the session; TRUE = hosting
        - ***name*** - optional name for listener

- **gfNetVoice::~gfNetVoice () `[virtual]`**
    - Destructor.
    - Function: Destructor
    - Purpose: Destroys gfNetVoice

4. Member Function Documentation

- **int gfNetVoice::GetNumVoices () `[inline]`**
    - Gets the number of current voices connected.

- **void gfNetVoice::GetVoicePosition (sgVec3 *pos*, int *Idx*)**
    - GetVoicePosition - gets the position of the voice.
    - Function: GetVoicePosition
    - Purpose: Gets the position for the specified voice
    - **Parameters:**
        - ***pos*** sgVec3 position to return with position values
        - ***Idx*** index value in array of **VOICE_INFO** objects

- **bool gfNetVoice::IsExcluded (int *voice*) `[inline]`**
    - IsExcluded - indicates whether voice is excluded.

- **bool gfNetVoice::IsObstructed (int *voice*) `[inline]`**
    - IsObstructed - indicates whether voice is obsructed.

- **bool gfNetVoice::IsOccluded (int *voice*) `[inline]`**
    - IsOccluded - indicates whether voice is occluded.

- **void gfNetVoice::RemExclusion (int *Idx*)**
    - RemExclusion - removes exclusion from this sound.
    - Function: RemExclusion
    - Purpose: Removes exclusion from this sound - sets default values

- **void gfNetVoice::RemObstruction (int *Idx*)**
    - RemObstruction - removes obstruction from this sound.
    - Function: RemObstruction
    - Purpose: Removes obstruction from this sound - sets default values

- **void gfNetVoice::RemOcclusion (int *Idx*)**
    - RemOcclusion - removes occlusion from the specified voice.
    - Function: RemOcclusion
    - Purpose: Removes occlusion from this sound - sets default values
    - **Parameters:**
        - ***voiceNum*** index of voice in array of **VOICE_INFO**

- **void gfNetVoice::SetExclusion (int *Idx*, long *exclusion*, float *exclusionLF*)**
    - SetExclusion - sets exclusion values for this sound.
    - Function: SetExclusion
    - Purpose: sets exclusion values for specified voice
    - **Parameters:**
        - ***Idx*** index to the voice in array of **VOICE_INFO** structs
        - ***exclusion*** - the exclusion value for EAX; range [-10000, 0] -10000 excludes sound to barely audible; 0 provides for no exclusion
        - ***exclusionLF*** - ratio of low to high frequency attenuation; range [0.0, 1.0] 0.0 indicates no attenuation at low freq; 1.0 indicates identical low and high freq attenuation Purpose: Sets the specified voice's exclusion settings

- **void gfNetVoice::SetMaxVoiceDistance (float *dist*)**
    - SetMaxVoiceDistance - Sets the maximum voice distance - distance at which no further attenuation occurs.
    - Function: SetMaxVoiceDistance
    - Purpose: Sets the maximum voice distance - distance at which no further attenuation occurs Default is 1 billion, virtually ensuring continuous attenuation
    - **Parameters:**
        - ***value*** - distance at which no further attenuation occurs

- **void gfNetVoice::SetMinVoiceDistance (float *dist*)**
    - SetMinVoiceDistance - Sets the distance at which no further gain is applied moving towards the voice object.
    - Function: SetMinVoiceDistance
    - Purpose: Sets the distance at which no further gain is applied moving towards the voice object Ex: if minimum voice distance set to 10, from 0.0 to 10.0 voice intensity will be constant
    - **Parameters:**
        - ***value*** - the minimum voice distance

- **void gfNetVoice::SetObstruction (int *Idx*, long *obstruction*, float *obstructionLF*)**
    - SetObstruction - sets obstruction values for this sound.
    - Function: SetObstruction
    - Purpose: Sets this voice's obstruction settings
    - **Parameters:**
        - ***Idx*** index to the voice in array of **VOICE_INFO** structs
        - ***obstruction*** - the obstruction value for EAX; range [-10000, 0] -10000 obstructs sound to barely audible; 0 provides for no obstruction
        - ***obstructionLF*** - ratio of low to high frequency attenuation; range [0.0, 1.0] 0.0 indicates no attenuation at low freq; 1.0 indicates identical low and high freq attenuation

- **void gfNetVoice::SetOcclusion (int *Idx*, long *occlusion*, float *occlusionLF*, float *occlusionRoomRatio*)**
    - SetOcclusionSettings - sets the specified voice to be occluded.
    - Function: SetOcclusion
    - Purpose: Sets this sound's occlusion settings
    - **Parameters:**
        - ***Idx*** - the index of the **VOICE_INFO** to modify occlusion
        - ***occlusion*** - the occlusion value for EAX; range [-10000, 0] -10000 occludes sound to barely audible; 0 provides for no occlusion
        - ***occlusionLF*** - ratio of low to high frequency attenuation; range [0.0, 1.0] 0.0 indicates no attenuation at low freq; 1.0 indicates identical low and high freq attenuation
        - ***occlusionRoomRatio*** - amount of occlusion to apply to non-direct path sound (reflections, reverberation) range [0.0, 10.0] 0.0 applies no additional occlusion to reflected/reverberated sounds; 10.0 (maximum) applies 10 times normal occlusion to non-direct path sound

The documentation for this class was generated from the following files:

- **gfnetvoice.h**
- **gfnetvoice.cpp**

H. GFSHAPE CLASS REFERENCE

`#include <gfaudioenvironment.h>`

1. Public Methods

- **gfShape ()**

Constructor.

- **~gfShape ()**

Destructor.

- **void** SetSize **(float size=7.5)**

SetSize - sets the "room" size for the environment.

- **float** GetSize **()**

GetSize - gets the size of the "room".

- **virtual void** SetLocation **(sgVec3 pos)=0**

SetLocation - sets the location for the shape.

- **virtual void** GetCenter **(sgVec3 pos)=0**

GetCenter - gets the center position of the shape.

- **virtual bool** Contains **(sgVec3 pos)=0**

Contains - determines whether position inside shape (pure virtual function).

2. Public Attributes

- **GZ_DECLARE_TYPE_INTERFACE**

3. Protected Attributes

- **float** mSize

4. Detailed Description

Class: gfShape Function: base class for all audio shapes

5. Constructor and Destructor Documentation

- **gfShape::gfShape ()**
    - Constructor.
    - Function: Constructor
    - Purpose: creates new gfShape

- **gfShape::~gfShape ()**
    - Destructor.
    - Function: Destructor
    - Purpose: destroys gfShape

6. Member Function Documentation

- **virtual bool gfShape::Contains (sgVec3 *pos*) `[pure virtual]`**
    - Contains - determines whether position inside shape (pure virtual function).
    - Implemented in **gfCube** and **gfSphere**.

- **virtual void gfShape::GetCenter (sgVec3 *pos*) `[pure virtual]`**
    - GetCenter - gets the center position of the shape.
    - Implemented in **gfCube** and **gfSphere**.

- **float gfShape::GetSize () `[inline]`**
    - GetSize - gets the size of the "room".

- **virtual void gfShape::SetLocation (sgVec3 *pos*) `[pure virtual]`**
    - SetLocation - sets the location for the shape.
    - Implemented in **gfCube**, and **gfSphere**.

- **void gfShape::SetSize (float *size* = 7.5)**
    - SetSize - sets the "room" size for the environment.
    - Function: SetSize
    - Purpose: sets the size of the shape for EAX purposes
    - **Parameters:**
        - ***size*** - the size of the shape

7. Member Data Documentation

- **gfShape::GZ_DECLARE_TYPE_INTERFACE**
    - Reimplemented in **gfCube a**nd **gfSphere**.

- **float gfShape::mSize `[protected]`**

The documentation for this class was generated from the following files:

- **gfaudioenvironment.h**
- **gfaudioenvironment.cpp**

I. GFSOUNDOBJECT CLASS REFERENCE

`#include <gfSoundObject.h>`

1. Public Types

- **enum gfSpatialEnum { GF_3D = 0, GF_2D }**

2. Public Methods

- gfSoundObject **(const char *filename, const char *name=0, bool networked=false,** gfSpatialEnum **type=GF_3D)**

Constructor.

- **~gfSoundObject ()**

Destructor - stops play and destroys this gfSoundObject.

- **const char *** GetFileName **()**

GetFileName - returns the filename of the sound.

- **void** Play **(bool loop=false)**

Play - plays wave file; if loop = TRUE, then loop wav continuously.

- **bool** IsPlaying **()**

IsPlaying - TRUE if this sound is playing.

- **bool** IsLooping **()**

IsLooping - TRUE if this sound is looping.

- **void** Stop **()**

Stop - stops wave file.

- **void** IncreasePitch **(float value)**

IncreasePitch - increases pitch (frequency) of playing wave file.

- **void** DecreasePitch **(float value)**

DecreasePitch - decreases pitch (frequency) of playing wave file.

- **void** SetPitch **(float pitch)**

SetPitch - sets the frequency (pitch) of wave file.

- **void ResetPitch ()**

ResetPitch - sets the frequency (pitch) of wave file to its original recorded frequency.

- **void** SetPan **(long pan)**

SetPan - sets the pan level for a 2D audio source.

- **void** IncreaseVolume **(float value)**

IncreaseVolume - increases gain, up to maximum level of original recording.

- **void** DecreaseVolume **(float value)**

DecreaseVolume - decreases gain, lower limit is 0.0 (mute).

- **void** SetVolume **(float volume)**

SetVolume - sets the gain for the sound object.

- **void** Position **(gfPosition *pos)**

Position - positions the sound object.

- **void** Position **(sgVec3 pos)**

Position - positions the sound object.

- **void SetRelative ()**

SetRelative - sets this sound object relative to the listener.

- **void SetNotRelative ()**

SetNotRelative - releases this sound object from relative positioning.

- **bool IsRelative ()**

IsRelative - returns whether sound is positioned relative to listener or world; TRUE = listener.

- **void** SetMinDistance **(float value=1.0f)**

SetMinDistance - distance away from sound object at which gain is clamped as you move closer to listener.

- **void** SetMaxDistance **(float value=1000000000.0f)**

SetMaxDistance - distance away from sound object at which gain does not further attenuate.

- **void** SetTether **(gfDynamic *obj)**

SetTether - sets this sound to tether to a gfDynamic derived class.

- **void** SetConeDirection **(float x, float y, float z)**

SetConeDirection - sets directivity.

- **void** SetConeAngles **(float, float)**

SetConeAngles - sets the width of the cone when sound is directional.

- **void** SetConeOutsideVolume **(float value)**

SetConeOutsideVolume - sets intensity (gain) of sound outside directional cone.

- **void** SetOcclusion **(long occlusion, float occlusionLF, float occlusionRoomRatio)**

SetOcclusion - sets occlusion values for this sound.

- **void RemOcclusion ()**

RemOcclusion - removes occlusion from this sound.

- **bool IsOccluded ()**

IsOccluded - TRUE indicates sound is occluded.

- **void** SetObstruction **(long obstruction, float obstructionLF)**

SetObstruction - sets obstruction values for this sound.

- **void RemObstruction ()**

RemObstruction - removes obstruction from this sound.

- **bool IsObstructed ()**

IsObstructed - TRUE indicates sound is obstructed.

- **void** SetExclusion **(long exclusion, float exclusionLF)**

SetExclusion - sets exclusion values for this sound.

- **void RemExclusion ()**

RemExclusion - removes exclusion from this sound.

- **bool IsExcluded ()**

IsExcluded - TRUE indicates sound is excluded.

- **void** SetNetworked **(bool network)**

SetNetworked - sets whether this sound object is networked to remote clients.

3. Public Attributes

- **GZ_DECLARE_TYPE_INTERFACE**

4. Protected Methods

- **void** Config **()**

Config - configures sound object.

- **bool ObtainResources ()**

ObtainResources - creates buffers.

- **void ReleaseResources ()**

Release - releases resources.

- **void CheckPlayingSounds ()**

CheckPlayingSounds - checks to ensure all sounds that have buffers are playing.

- **int CreateNewSoundObject ()**

CreateNewSoundObject - internal management and tracking of this sound object.

- **bool OpenWaveFile ()**

OpenWaveFile - mmio reading method.

- **void** ResetFile **()**

ResetFile - mmio reset method.

- **void InitialBufferLoad ()**

InitialBufferLoad - reads wave file into memory.

- **gzVoid** onNotify **(gzNotifyMessage *message)**

onNotify - internal messaging.

- **bool SetOcclusionSettings ()**

SetOcclusionSettings.

- **bool SetObstructionSettings ()**

SetObstructionSettings.

- **bool SetExclusionSettings ()**

SetExclusionSettings.

- **void** Send **(gfSoundActionEnum action)**

Send - sends gfSoundActionPacket to network.

5. Protected Attributes

- **LPDIRECTSOUNDBUFFER** pDSB
- **LPDIRECTSOUND3DBUFFER** pDSB3D
- **LPKSPROPERTYSET** pEAXSource
- **gfSpatialEnum mType**
- **gzRefPointer< gfDynamic >** mTether
- **unsigned long** mPitch
- **long mVolume**
- **long** mPan
- **float mMinDistance**
- **float mMaxDistance**
- **float mMinConeAngle**
- **float mMaxConeAngle**
- **float mOuterConeVolume**
- **sgVec3 mConeDirection**
- **char** m_Filename **[128]**
- **HMMIO m_hMmioFile**
- **DSBUFFERDESC** m_DSBufDesc
- **UINT BytesToEndOfFile**
- **UINT DataChunkSize**
- **UINT DataChunkOffset**
- **bool isConfigured**
- **bool** mPlay
- **bool mLooping**
- **long** mOcc
- **float** mOccLF
- **float mOccRoomRatio**
- **long** mObstr
- **float mObstrLF**
- **long** mExcl
- **float** mExclLF
- **bool eaxSupported**
- **bool** mRel
- **bool isObstructed**
- **bool isOccluded**
- **bool isExcluded**
- **bool isNetworked**

6. Detailed Description

Class: gfSoundObject

Function: class to represent a 2D or 3D sound source. 2D capabilities: volume control, pitch control, pans left/right 3D capabilities: distance attenuation, velocity (Doppler) positioning, pitch, volume, and environmental effects.

- **Can be linked to a visual object or stand alone**
- **gfSoundObjects have occlusion, obstruction, and exclusion settings that are specific to the sound loaded when 3D NOTE: Most sound cards are limited in the number of 3D and 2D hardware buffers - for sounds that do not require spatialization, utilize 2D settings**

7. Member Enumeration Documentation

- **enum gfSoundObject::gfSpatialEnum**
  - **Enumeration values:**
    - ***GF_3D***
    - ***GF_2D***

8. Constructor and Destructor Documentation

- **gfSoundObject::gfSoundObject (const char * *filename*, const char * *name* = 0, bool *networked* = false, gfSpatialEnum *type* = GF_3D)**
  - Constructor.
  - Function: Constructor
  - Purpose: creates new gfSoundObject
  - **Parameters:**
    - ***name*** - optional name for sound object
- **gfSoundObject::~gfSoundObject ()**
  - Destructor - stops play and destroys this gfSoundObject.
  - Function: Destructor
  - Purpose: Destroy this sound object

9. Member Function Documentation

- **void gfSoundObject::CheckPlayingSounds () `[protected]`**
  - CheckPlayingSounds - checks to ensure all sounds that have buffers are playing.
  - Function: OpenWaveFile
  - Purpose: opens wave file header for reading file information

- **void gfSoundObject::Config (void) `[protected]`**
  - Config - configures sound object.
  - Function: Config
  - Purpose: configures variables and DirectSound objects

- **int gfSoundObject::CreateNewSoundObject () `[protected]`**
  - CreateNewSoundObject - internal management and tracking of this sound object.
  - Function: CreateNewSoundObject

- Purpose: creates new gfSoundObject and stores it in global sound object list
- **Parameters:**
  - ***int*** - integer representing number of sound objects

- **void gfSoundObject::DecreasePitch (float *amount*)**
  - DecreasePitch - decreases pitch (frequency) of playing wave file.
  - Function: DecreasePitch
  - Purpose: Decreases the pitch (frequency) of this sound object by the given value
  - **Parameters:**
    - ***value*** - the amount of the pitch decrease

- **void gfSoundObject::DecreaseVolume (float *amount*)**
  - DecreaseVolume - decreases gain, lower limit is 0.0 (mute).
  - Function: DecreaseVolume
  - Purpose: Decreases the volume (intensity) level by the given value
  - **Parameters:**
    - ***value*** - the amount of volume (intensity) to decrease

- **const char* gfSoundObject::GetFileName () `[inline]`**
  - GetFileName - returns the filename of the sound.

- **void gfSoundObject::IncreasePitch (float *amount*)**
  - IncreasePitch - increases pitch (frequency) of playing wave file.
  - Function: IncreasePitch
  - Purpose: Increases the pitch (frequency) of this sound object by the given value
  - **Parameters:**
    - ***value*** - the amount of the pitch increase

- **void gfSoundObject::IncreaseVolume (float *amount*)**
  - IncreaseVolume - increases gain, up to maximum level of original recording.
  - Function: IncreaseVolume
  - Purpose: Increases the volume (intensity) level by the given value
  - **Parameters:**
    - ***value*** - the amount of volume (intensity) to increase

- **void gfSoundObject::InitialBufferLoad () `[protected]`**
  - InitialBufferLoad - reads wave file into memory.
  - Function: Constructor
  - Purpose: creates new **gfListener**
  - **Parameters:**
    - ***name*** - optional name for listener

- **bool gfSoundObject::IsExcluded () `[inline]`**
  - IsExcluded - TRUE indicates sound is excluded.

- **bool gfSoundObject::IsLooping ()**

- IsLooping - TRUE if this sound is looping.
- Function: IsLooping
- Purpose: indicate whether this sound object is looping
- **Returns:**
  - bool TRUE if looping; FALSE if not

- **bool gfSoundObject::IsObstructed () `[inline]`**
  - IsObstructed - TRUE indicates sound is obstructed.

- **bool gfSoundObject::IsOccluded () `[inline]`**
  - IsOccluded - TRUE indicates sound is occluded.

- **bool gfSoundObject::IsPlaying ()**
  - IsPlaying - TRUE if this sound is playing.
  - Function: IsPlaying
  - Purpose: indicate whether this sound object is playing
  - **Returns:**
    - bool TRUE if playing; FALSE if not

- **bool gfSoundObject::IsRelative () `[inline]`**
  - IsRelative - returns whether sound is positioned relative to listener or world; TRUE = listener.

- **bool gfSoundObject::ObtainResources () `[protected]`**
  - ObtainResources - creates buffers.
  - Function: ObtainResources
  - Purpose: configures DirectSound derived objects

- **gzVoid gfSoundObject::onNotify (gzNotifyMessage * *message*) `[protected]`**
  - onNotify - internal messaging.
  - Function: onNotify
  - Purpose: Internal message handler
  - **Parameters:**
    - ***message*** - Gizmo3D message struct - used internally

- **bool gfSoundObject::OpenWaveFile () `[protected]`**
  - OpenWaveFile - mmio reading method.
  - Function: OpenWaveFile
  - Purpose: opens wave file header for reading file information

- **void gfSoundObject::Play (bool *loop* = false)**
  - Play - plays wave file; if loop = TRUE, then loop wav continuously.
  - Function: Play
  - Purpose: Plays this sound object

- **void gfSoundObject::Position (sgVec3 *pos*)**
    - Position - positions the sound object.
    - Function: Position
    - Purpose: Position this sound object
    - **Parameters:**
        - ***pos*** - sgVec for new position of sound object
- **void gfSoundObject::Position (gfPosition * *pos*)**
    - Position - positions the sound object.
    - Function: Position
    - Purpose: Position this sound object
    - **Parameters:**
        - ***pos*** - gfPosition for new position of sound object

- **void gfSoundObject::ReleaseResources () `[protected]`**
    - Release - releases resources.
    - Function: Release
    - Purpose: Releases resources for this sound object

- **void gfSoundObject::RemExclusion ()**
    - RemExclusion - removes exclusion from this sound.
    - Function: RemExclusion
    - Purpose: Removes exclusion from this sound - sets default values

- **void gfSoundObject::RemObstruction ()**
    - RemObstruction - removes obstruction from this sound.
    - Function: RemObstruction
    - Purpose: Removes obstruction from this sound - sets default values

- **void gfSoundObject::RemOcclusion ()**
    - RemOcclusion - removes occlusion from this sound.
    - Function: RemOcclusion
    - Purpose: Removes occlusion from this sound - sets default values

- **void gfSoundObject::ResetFile () `[protected]`**
    - ResetFile - mmio reset method.
    - Function: ResetFile
    - Purpose: resets the mmio read function for reading wave data after header determination

- **void gfSoundObject::ResetPitch ()**
    - ResetPitch - sets the frequency (pitch) of wave file to its original recorded frequency.

- Function: ResetPitch
- Purpose: Sets the frequency (pitch) of this sound object to its original recorded frequency

- **void gfSoundObject::Send (gfSoundActionEnum *action*) `[protected]`**
  - Send - sends gfSoundActionPacket to network.
  - Function: Send
  - Purpose: Sets the sound to automatically transmit play, stop calls to the network
  - **Parameters:**
    - ***network*** TRUE = transmit automatically; FALSE = no transmission

- **void gfSoundObject::SetConeAngles (float *inner*, float *outer*)**
  - SetConeAngles - sets the width of the cone when sound is directional.
  - Function: SetConeAngles
  - Purpose: sets the angular measurement for inner and outer cones directivity scenario
  - **Parameters:**
    - ***inner*** the inner angle
    - ***outer*** the outer angle

- **void gfSoundObject::SetConeDirection (float *x*, float *y*, float *z*)**
  - SetConeDirection - sets directivity.
  - Function: SetConeDirection
  - Purpose: sets the directivity of the cone
  - **Parameters:**
    - ***x*** - look at x direction
    - ***x*** - look at y direction
    - ***x*** - look at z direction

- **void gfSoundObject::SetConeOutsideVolume (float *vol*)**
  - SetConeOutsideVolume - sets intensity (gain) of sound outside directional cone.
  - Function: SetConeOutsideVolume
  - Purpose: Sets the outside cone volume Outside cone volume is the attenuation applied outside the cone of directivity
  - **Parameters:**
    - ***value*** - gain value for the volume outside the directed cone; 0 = no volume range = (0.0f, inf)

- **void gfSoundObject::SetExclusion (long *exclusion*, float *exclusionLF*)**
  - SetExclusion - sets exclusion values for this sound.
  - Function: SetExclusion
  - Purpose: Sets this sound's exclusion settings
  - **Parameters:**
    - ***exclusion*** - the exclusion value for EAX; range [-10000, 0] -10000 excludes sound to barely audible; 0 provides for no exclusion
    - ***exclusionLF*** - ratio of low to high frequency attenuation; range [0.0, 1.0] 0.0 indicates no attenuation at low freq; 1.0 indicates identical low and high freq attenuation

- **bool gfSoundObject::SetExclusionSettings () `[protected]`**
    - SetExclusionSettings.
    - Function: SetExclusionSettings
    - Purpose: Sets the respective exclusion values into the EAX property set
    - **Returns:**
        - bool TRUE = exclusion values successfully set

- **void gfSoundObject::SetMaxDistance (float *distance* = 1000000000.0f)**
    - MaxDistance - distance away from sound object at which gain does not further attenuate.
    - Function: MaxDistance
    - Purpose: Sets the distance at which no further attenuation is applied moving away from sound object
    - **Parameters:**
        - ***value*** - the maximum distance - defaults to 1 billion

- **void gfSoundObject::SetMinDistance (float *distance* = 1.0f)**
    - MinDistance - distance away from sound object at which gain is clamped as you move closer to listener.
    - Function: MinDistance
    - Purpose: Sets the distance at which no further gain is applied moving towards the sound object
    - **Parameters:**
        - ***value*** - the minimum distance - defaults to 1.0f

- **void gfSoundObject::SetNetworked (bool *network*)**
    - SetNetworked - sets whether this sound object is networked to remote clients.
    - Function: SetNetworked
    - Purpose: Sets the sound to automatically transmit play, stop calls to the network
    - **Parameters:**
        - ***network*** TRUE = transmit automatically; FALSE = no transmission

- **void gfSoundObject::SetNotRelative ()**
    - RemoveRelative - releases this sound object from relative positioning.
    - Function: RemoveRelative
    - Purpose: Sets this sound object to be positioning globally; all subsequent calls to **Position**() will place this sound object in global coordinates

- **void gfSoundObject::SetObstruction (long *obstruction*, float *obstructionLF*)**
    - SetObstruction - sets obstruction values for this sound.
    - Function: SetObstruction
    - Purpose: Sets this sound's obstruction settings
    - **Parameters:**
        - ***obstruction*** - the obstruction value for EAX; range [-10000, 0] -10000 obstructs sound to barely audible; 0 provides for no obstruction

        - ***obstructionLF*** - ratio of low to high frequency attenuation; range [0.0, 1.0] 0.0 indicates no attenuation at low freq; 1.0 indicates identical low and high freq attenuation

- **bool gfSoundObject::SetObstructionSettings () `[protected]`**
    - SetObstructionSettings.
    - Function: SetOcclusion
    - Purpose: Sets the respective occlusion values into the EAX property set
    - **Returns:**
        - bool TRUE = obstruction values successfully set

- **void gfSoundObject::SetOcclusion (long *occlusion*, float *occlusionLF*, float *occlusionRoomRatio*)**
    - SetOcclusion - sets occlusion values for this sound.
    - Function: SetOcclusion
    - Purpose: Sets this sound's occlusion settings
    - **Parameters:**
        - ***occlusion*** - the occlusion value for EAX; range [-10000, 0] -10000 occludes sound to barely audible; 0 provides for no occlusion
        - ***occlusionLF*** - ratio of low to high frequency attenuation; range [0.0, 1.0] 0.0 indicates no attenuation at low freq; 1.0 indicates identical low and high freq attenuation
        - ***occlusionRoomRatio*** - amount of occlusion to apply to non-direct path sound (reflections, reverberation) range [0.0, 10.0] 0.0 applies no additional occlusion to reflected/reverberated sounds; 10.0 (maximum) applies 10 times normal occlusion to non-direct path sound

- **bool gfSoundObject::SetOcclusionSettings () `[protected]`**
    - SetOcclusionSettings.
    - Function: SetOcclusionSettings
    - Purpose: Sets the respective occlusion values into the EAX property set
    - **Returns:**
        - bool TRUE = occlusion settings successfully completed

- **void gfSoundObject::SetPan (long *pan*)**
    - SetPan - sets the pan level for a 2D audio source.
    - Function: SetPan
    - Purpose: sets the pan left or right for a 2D audio source
    - **Parameters:**
        - ***pan*** - the amount of the pan The value in pan is measured in hundredths of a decibel (dB), in the range of DSBPAN_LEFT to DSBPAN_RIGHT. These values are currently defined in Dsound.h as –10,000 and 10,000 respectively. The value DSBPAN_LEFT means the right channel is attenuated by 100 dB. The value DSBPAN_RIGHT means the left channel is attenuated by 100 dB. The neutral value is DSBPAN_CENTER, defined as zero. This value of 0 in the pan parameter means that both channels are at full volume (they are attenuated by 0 decibels). At any setting other than DSBPAN_CENTER, one of the channels is at full volume and the other is attenuated.

- **void gfSoundObject::SetPitch (float *pitch*)**
    - SetPitch - sets the frequency (pitch) of wave file.
    - Function: SetPitch
    - Purpose: Sets the frequency (pitch) of this sound object; range (0.0f, inf] A value of DSBFREQUENCY_ORIGINAL resets back to the original value.
    - **Parameters:**
        - ***pitch*** - the frequency (pitch) to set; range (100.0, 100,000.0)

- **void gfSoundObject::SetRelative ()**
    - SetRelative - sets this sound object relative to the listener.
    - Function: SetRelative
    - Purpose: Sets this sound object to be positioning relative to **gfListener**; all subsequent calls to **Position**() will place this sound object in relative positions to the one listener in the given context

- **void gfSoundObject::SetTether (gfDynamic * *obj*)**
    - SetTethered - sets this sound to tether to a gfDynamic derived class.
    - Function: SetTether
    - Purpose: Sets the gfDynamic derived class this gfSoundObject is tethered to for positioning
    - **Parameters:**
        - ***obj*** - pointer to a gfDynamic derived class to tether to

- **void gfSoundObject::SetVolume (float *volume*)**
    - SetVolume - sets the gain for the sound object.
    - Function: SetVolume
    - Purpose: set the volume of the sound object. The volume is specified in hundredths of decibels (dB). Allowable values are between DSBVOLUME_MAX (no attenuation) and DSBVOLUME_MIN (silence). These values are currently defined in Dsound.h as 0 and -10,000 respectively. The value DSBVOLUME_MAX represents the original, unadjusted volume of the stream. The value DSBVOLUME_MIN indicates an audio volume attenuated by 100 dB, which, for all practical purposes, is silence. Currently DirectSound does not support amplification.
    - **Parameters:**
        - ***volume*** the amount of volume (GAIN)

- **void gfSoundObject::Stop ()**
    - Stop - stops wave file.
    - Function: Stop Purpose: Stops playing on this sound object

10. Member Data Documentation

- **UINT gfSoundObject::BytesToEndOfFile `[protected]`**
    - buffer description = bytes to the end of the file
- **UINT gfSoundObject::DataChunkOffset `[protected]`**
    - number of bytes in offset
- **UINT gfSoundObject::DataChunkSize `[protected]`**

    - number of bytes in chunk
- **bool gfSoundObject::eaxSupported [protected]**
    - whether EAX is supported
- **gfSoundObject::GZ_DECLARE_TYPE_INTERFACE**
- **bool gfSoundObject::isConfigured [protected]**
    - whether sound object is configured
- **bool gfSoundObject::isExcluded [protected]**
    - indicates whether sound is excluded
- **bool gfSoundObject::isNetworked [protected]**
    - indicates whether sound is networked
- **bool gfSoundObject::isObstructed [protected]**
    - indicates whether sound is obstructed
- **bool gfSoundObject::isOccluded [protected]**
    - indicates whether sound is occluded
- **DSBUFFERDESC gfSoundObject::m_DSBufDesc [protected]**
    - buffer description
- **char gfSoundObject::m_Filename[128] [protected]**
    - file name for the wave file
- **HMMIO gfSoundObject::m_hMmioFile [protected]**
    - mmio file used to read wave file
- **sgVec3 gfSoundObject::mConeDirection [protected]**
    - cone direction
- **long gfSoundObject::mExcl [protected]**
    - exclusion setting
- **float gfSoundObject::mExclLF [protected]**
    - exclusion setting for low frequencies
- **bool gfSoundObject::mLooping [protected]**
    - indicates sound is looping
- **float gfSoundObject::mMaxConeAngle [protected]**
    - maximum cone angle
- **float gfSoundObject::mMaxDistance [protected]**
    - maximum distance
- **float gfSoundObject::mMinConeAngle [protected]**
    - minimum cone angle
- **float gfSoundObject::mMinDistance [protected]**
    - min distance
- **long gfSoundObject::mObstr [protected]**
    - obstruction setting
- **float gfSoundObject::mObstrLF [protected]**
    - obstruction setting for low frequencies
- **long gfSoundObject::mOcc [protected]**
    - occlusion setting
- **float gfSoundObject::mOccLF [protected]**
    - occlusion setting for low frequencies

- **float gfSoundObject::mOccRoomRatio [protected]**
  - occlusion room ratio setting
- **float gfSoundObject::mOuterConeVolume [protected]**
  - volume setting for outside cone areas
- **long gfSoundObject::mPan [protected]**
  - stored pan value
- **unsigned long gfSoundObject::mPitch [protected]**
  - stored pitch value
- **bool gfSoundObject::mPlay [protected]**
  - indicates gfSoundObject is playing or Play() called prior to config()
- **bool gfSoundObject::mRel [protected]**
  - indicates sound is relative to listener
- **gzRefPointer<gfDynamic> gfSoundObject::mTether [protected]**
  - gfObject to tether this sound to for positioning
- **gfSpatialEnum gfSoundObject::mType [protected]**
  - type of sound object - either GF_2D or GF_3D
- **long gfSoundObject::mVolume [protected]**
  - stored volume value
- **LPDIRECTSOUNDBUFFER gfSoundObject::pDSB [protected]**
  - secondary sound buffer
- **LPDIRECTSOUND3DBUFFER gfSoundObject::pDSB3D [protected]**
  - DirectSound3D buffer - for 3D positioning
- **LPKSPROPERTYSET gfSoundObject::pEAXSource [protected]**
  - EAX property set interface

The documentation for this class was generated from the following files:

- **gfSoundObject.h**
- **gfSoundObject.cpp**

J. GFSPHERE CLASS REFERENCE

`#include <gfaudioenvironment.h>`

1. Public Methods

- gfSphere **(float x=0.0f, float y=0.0f, float z=0.0f, float radius=1.0f)**

Constructor - optional args for sphere center and radius.

- **~gfSphere ()**

Destructor.

- **void** SetSphere **(float x, float y, float z, float radius)**

SetSphere - sets environment sphere.

- **virtual void** SetLocation **(sgVec3 pos)**

SetLocation - sets the location for the shape.

- **bool** Contains **(sgVec3 pos)**

Contains - determines whether position inside sphere.

- **void** GetCenter **(sgVec3 pos)**

GetCenter - gets the center point of the sphere.

2. Public Attributes

- **GZ_DECLARE_TYPE_INTERFACE**

3. Detailed Description

Class: gfSphere
Function: class for sphere audio shape

4. Constructor and Destructor Documentation

- **gfSphere::gfSphere (float *x* = 0.0f, float *y* = 0.0f, float *z* = 0.0f, float *radius* = 1.0f)**
    - Constructor - optional args for sphere center and radius.
    - Function: Constructor
    - Purpose: creates new gfSphere
    - **Parameters:**
        - ***x*** - sphere center x coordinate
        - ***y*** - sphere center y coordinate
        - ***z*** - sphere center z coordinate
        - ***radius*** - sphere radius

- **gfSphere::~gfSphere ()**
    - Destructor.
    - Function: Destructor
    - Purpose: destroys gfSphere

5. Member Function Documentation

- **bool gfSphere::Contains (sgVec3 *pos*) `[virtual]`**
    - Contains - determines whether position inside sphere.
    - Function: Contains
    - Purpose: determine whether the gfPosition is contained in the shape
    - **Parameters:**
        - ***pos*** - the position to check for containment
    - **Returns:**
        - bool - TRUE = inside shape; FALSE = outside shape
    - Implements **gfShape**.

- **void gfSphere::GetCenter (sgVec3 *pos*) `[virtual]`**
    - GetCenter - gets the center point of the sphere.
    - Function: GetCenter
    - Purpose: determine the center point of the cube
    - **Parameters:**
        - ***pos*** - the position to fill in data
    - Implements **gfShape**).

- **void gfSphere::SetLocation (sgVec3 *pos*) `[virtual]`**
    - SetLocation - sets the location for the shape.
    - Function: SetLocation
    - Purpose: sets the center location for the sphere
    - **Parameters:**
        - ***pos*** - the position of the center of the sphere
    - Implements **gfShape**.

- **void gfSphere::SetSphere (float *x*, float *y*, float *z*, float *radius*)**
    - SetSphere - sets environment sphere.
    - Function: SetSphere
    - Purpose: sets the shape of the sphere; size and radius
    - **Parameters:**
        - ***x*** - sphere center x coordinate
        - ***y*** - sphere center y coordinate
        - ***z*** - sphere center z coordinate
        - ***radius*** - sphere radius

6. Member Data Documentation

- **gfSphere::GZ_DECLARE_TYPE_INTERFACE**

    - Reimplemented from **gfShape**.

The documentation for this class was generated from the following files:

- **gfaudioenvironment.h**
- **gfaudioenvironment.cpp**

K. VOICE_INFO STRUCT REFERENCE

`#include <gfnetvoice.h>`

1. Public Attributes

- **DWORD** id
- **LPDIRECTSOUND3DBUFFER** pDSB3D
- **LPKSPROPERTYSET** pEAXvoice
- **long** mOcc
- **float** mOccLF

- **float mOccRoomRatio**
- **bool isOccluded**
- **long** mObstr
- **float mObstrLF**
- **bool isObstructed**
- **long** mExcl
- **float** mExclLF
- **bool isExcluded**
- **float** mX
- **float** mY
- **float** mZ

2. Detailed Description

Struct: VOICE_INFO

Function: struct to store information about each remote voice client. Contains necessary DirectVoice buffers for 3D spatialization Permits setting of occlusion, obstruction or exclusion for individual voices.

3. Member Data Documentation

- **DWORD VOICE_INFO::id**
- **bool VOICE_INFO::isExcluded**
    - indicates whether voice is excluded
- **bool VOICE_INFO::isObstructed**
    - indicates whether voice is obstructed
- **bool VOICE_INFO::isOccluded**
    - indicates whether voice is occluded
- **long VOICE_INFO::mExcl**
    - voice exclusion setting
- **float VOICE_INFO::mExclLF**
    - voice exclusion setting for low frequencies
- **long VOICE_INFO::mObstr**
    - voice obstruction setting
- **float VOICE_INFO::mObstrLF**
    - voice obstruction setting for low frequencies
- **long VOICE_INFO::mOcc**
    - voice occlusion setting
- **float VOICE_INFO::mOccLF**
    - voice occlusion setting for low frequencies
- **float VOICE_INFO::mOccRoomRatio**
    - voice occlusion room ratio setting
- **float VOICE_INFO::mX**
    - voice x position
- **float VOICE_INFO::mY**
    - voice y position
- **float VOICE_INFO::mZ**

    - voice z position
- **LPDIRECTSOUND3DBUFFER VOICE_INFO::pDSB3D**
    - 3D sound buffer for voice
- **LPKSPROPERTYSET VOICE_INFO::pEAXvoice**
    - EAX property interface for voice

The documentation for this struct was generated from the following file:

- **gfnetvoice.h**
- **gfnetvoice.cpp**

APPENDIX B. AUSERVERLIB DOCUMENTATION

A. AUBASE CLASS REFERENCE

`#include <aubase.h>`

1. Public Methods

- **void** SendNotify **(char *message,** auRefData ***data)**

SendNotify - pass messages internally.

- **void** AddNotifier **(auBase *notifier)**

AddNotifier - registers auClass with caller.

- **bool** IsOfClass **(gzType *type) const**
- **bool** IsExactlyClass **(gzType *type) const**
- **virtual void** SetName **(const char *name)**

SetName - sets the name of this object.

- **virtual const char *** GetName **() const**
- **auBase ()**
- **virtual** ~auBase **()**

Destructor.

2. Public Attributes

- **GZ_DECLARE_TYPE_INTERFACE**

3. Protected Attributes

- **char** mName **[128]**

4. Detailed Description

Class: auBase Function:

Base class from which all other auClasses are derived from. Supplies basic naming, printing and class reference methods.

5. Constructor and Destructor Documentation

- **auBase::auBase () `[inline]`**
  - Construct a new auBase. auBase is a pure virtual object and therefore cannot be created on its own. Only derived classes which implement the virtual methods may be constructed.
- **virtual auBase::~auBase () `[inline, virtual]`**
  - Destructor.

6. Member Function Documentation

- **void auBase::AddNotifier (auBase * *notifier*)**
    - AddNotifier - registers auClass with caller.
    - Tell this object to subscribe to notifier's messages. Notifier must be an object derived from gfBase.
    - **See also:**
        - **SendNotify**()
    - **Parameters:**
        - ***notifier*** - The object to receive messages *from*

- **virtual const char* auBase::GetName () const `[inline, virtual]`**
    - GetName - Get the name of object

- **bool auBase::IsExactlyClass (gzType * *type*) const `[inline]`**
    - Is this instance exactly the passed class type? Get the gzType by calling getClassType()

- **bool auBase::IsOfClass (gzType * *type*) const `[inline]`**
    - Is this instance derived from the passed class type? Get the gzType by calling getClassType()

- **void auBase::SendNotify (char * *message*, auRefData * *data*)**
    - SendNotify - pass messages internally.
    - Send a message from this object to any other object that subscribed to this object. A text string and gfRefData can be sent in the message. Any subscriber to this object can then parse the text string and user data.
    - **Parameters:**
        - ***message*** - Text string to send in message
        - ***data*** - Any additional data that needs to be sent with this message
    - **See also:**
        - **AddNotifier**()

- **void auBase::SetName (const char * *name*) `[virtual]`**
    - SetName - sets the name of this object.

7. Member Data Documentation

- **auBase::GZ_DECLARE_TYPE_INTERFACE**
    - Reimplemented in **auChannel**, **auListener**, **auSound auSource**, and **auSystem**.

- **char auBase::mName[128] `[protected]`**
    - Reimplemented in **auSource**.

The documentation for this class was generated from the following files:

- **aubase.h**
- **aubase.cpp**

B. AUCHANNEL CLASS REFERENCE

`#include <auchannel.h>`

1. Public Methods

- auChannel **(int channel, const char *name=0)**

Constructor.

- **virtual** ~auChannel **()**

Destructor.

- **bool** SetChannel **(int channel)**

SetChannel - sets input ausim3D channel.

- **int** GetChannel **(void)**
- **bool LinkToListener (auListener *listener)**

*LinkToListener - links source to specified **auListener**.*

- **bool** UnLink **()**

Unlink - unlinks source to all auListeners.

- **bool SetExclusive (auListener *listener)**

SetExclusive - directs this channel exclusively to specified listener.

- **bool RemExclusive ()**

RemExclusive - removes any exclusivity of this source to any listener.

- **bool SetVolumeForListener (auListener *listener, float dB)**

SetVolumeForListener - sets volume to the specified listener.

2. Public Attributes

- **GZ_DECLARE_TYPE_INTERFACE**

3. Detailed Description

Class: auChannel

Function: Class representing all aspects of an ausim source generated from a live audio feed. Contains methods for setting/getting source position, radiation patterns, names, and source-specific rolloff. Contains methods to link/unlink to specified listener.

4. Constructor and Destructor Documentation

- **auChannel::auChannel (int *channel*, const char * *name* = 0)**
    - Constructor.
    - Function: Constructor
    - Purpose: Creates new channel
    - **Parameters:**
        - ***channel*** - channel number of this channel - corresponds to input on ausim3D
        - ***name*** - string for name of this channel

- **auChannel::~auChannel () `[virtual]`**
    - Destructor.
    - Function: Destructor
    - Purpose: Destroys this channel

5. Member Function Documentation

- **int auChannel::GetChannel (void) `[inline]`**
    - GetChannel - gets input ausim3D channel

- **bool auChannel::LinkToListener (auListener * *listener*)**
    - LinkToListener - links source to specified **auListener**.
    - Function: LinkToListener
    - Purpose: Link this source to the specified **auListener**
    - **Parameters:**
        - ***listener*** - the **auListener** to link with

- **bool auChannel::RemExclusive ()**
    - RemExclusive - removes any exclusivity of this source to any listener.
    - Function: RemExclusive
    - Purpose: removes any exclusivity setting from this sound

- **bool auChannel::SetChannel (int *channel*)**
    - SetChannel - sets input ausim3D channel.
    - Function: SetChannel
    - Purpose: Set the channel for this course
    - **Parameters:**
        - ***channel*** - the channel of this source

- **bool auChannel::SetExclusive (auListener * *listener*)**
    - SetExclusive - directs this channel exclusively to specified listener.
    - Function: SetExclusive
    - Purpose: sets this channel to only be heard by specified listener
    - **Parameters:**

        - ***listener*** - the exclusive **auListener** to be heard by

- **bool auChannel::SetVolumeForListener (auListener * *listener*, float *volume*)**
    - SetVolumeForListener - sets volume to the specified listener.
    - Function: SetVolumeForListener
    - Purpose: sets this channel's volume for a specified listener
    - **Parameters:**
        - ***listener*** - the **auListener**

- **bool auChannel::UnLink ()**
    - Unlink - unlinks source to all auListeners.
    - Function: LinkToListener
    - Purpose: Link this source to the specified **auListener**
    - **Parameters:**
        - ***listener*** - the **auListener** to link with

6. Member Data Documentation

- **auChannel::GZ_DECLARE_TYPE_INTERFACE**
    - Reimplemented from **auSource**.

The documentation for this class was generated from the following files:

- **auchannel.h**
- **auchannel.cpp**

C. AULIST CLASS REFERENCE

```
#include <aulist.h>
```

1. Public Methods

- auList **(int numElements=0)**

Constructor.

- **virtual** ~auList **()**

Destructor.

- **void** Rem **(**auBase ***data)**

Rem - removes object from list.

- **int** GetNum **(void)**

GetNum - gets the number of objects in the list.

- **void *** Get **(const int elementIdx)**

Get - gets the object at the specified index.

- **void** Add **(**auBase ***data)**

Add - adds an object to the end of the list.

- **void** InsertAt **(**auBase ***data, int elementIdx)**

InsertAt - inserts an object at the specified index in the list.

- **void** Clear **(void)**

Clear - removes all objects from the list.

2. Constructor and Destructor Documentation

- **auList::auList (int *numElements* = 0)**
  - Constructor.
  - Function: Constructor
  - Purpose: Creates new auList
  - **Parameters:**
    - ***numElements*** - number of initial slots for elements

- **auList::~auList () `[virtual]`**
  - Destructor.
  - Function: Destructor
  - Purpose: Destroys auList

3. Member Function Documentation

- **void auList::Add (auBase * *data*)**
  - Add - adds an object to the end of the list.
  - Function: Add
  - Purpose: Add an element to the list
  - **Parameters:**
    - ***data*** **auBase** derived object to add to the list

- **void auList::Clear (void)**
  - Clear - removes all objects from the list.
  - Function: Clear
  - Purpose: clears all elements from the list.

- **void * auList::Get (const int *elementIdx*)**
  - Get - gets the object at the specified index.
  - Function: Get
  - Purpose: Get one element from the list.
  - **Parameters:**
    - ***elementIdx*** index of the element to get
  - **Returns:**
    - pointer to the element at the specified index

- **int auList::GetNum (void)**
  - GetNum - gets the number of objects in the list.

    - Function: GetNum
    - Purpose: Get the number of elements in the list.
    - **Returns:**
        - number of elements in the list.

- **void auList::InsertAt (auBase * *data*, int *elementIdx*)**
    - InsertAt - inserts an object at the specified index in the list.
    - Function: InsertAt
    - Purpose: Inserts an element at the specified index
    - **Parameters:**
        - ***data*** **auBase**-derived object to insert
        - ***elementIdx*** index to insert at

- **void auList::Rem (auBase * *data*)**
    - Rem - removes object from list.
    - Function: Rem
    - Purpose: Remove an item from the list.
    - **Parameters:**
        - ***data*** The element to remove NOTE: This will not delete the memory, only remove a reference count from the data and take it out of the list. If the reference count of data is zero, it will delete itself.

The documentation for this class was generated from the following files:

- **aulist.h**
- **aulist.cpp**

D. AULISTENER CLASS REFERENCE

`#include <aulistener.h>`

1. Public Methods

- auListener **(const char *name=0)**

Constructor.

- **virtual** ~auListener **()**

Destructor.

- **bool** SetVoiceInputChannel **(int channel)**

SetVoiceInputChannel - sets the channel for voice input.

- **bool** SetHRTF **(const char *hrtfName)**

SetHRTF - sets user specified HRTF in ausim3D.

- **const char *** GetHRTF **() const**

- **bool SetPosition (auPosition *pos=NULL)**

SetPosition - updates position.

- **void GetPosition (auPosition *pos)**

GetPosition - returns current position.

- **int** GetID **() const**

- **bool SetMouthOffset (auPosition *pos)**

SetMouthOffset - sets offset for mouth position from ears.

- **bool SetMouthRadPattern (auRadPattern *pattern)**

SetMouthRadPattern - sets radiation pattern of mouth.

- **void GetMouthRadPattern (auRadPattern *pattern)**

GetMouthRadPattern - gets radiation pattern of mouth.

- **bool** SetMouthVolume **(float dB)**

SetMouthVolume - sets the overall volume of the mouth source.

- **bool** DecreaseMouthVolume **(float dBamount)**

DecreaseMouthVolume - decreases the mouth volume by the specified factor.

- **bool** IncreaseMouthVolume **(float dBamount)**

IncreaseMouthVolume - increases the mouth volume by the specified factor.

- **bool** SetExclusive **(auListener *listener)**

SetExclusive - sets this listener's voice exclusive to specified listener.

- **bool RemExclusive ()**

RemExclusive - removes the exclusivity of this listener to the specified listener.

- **bool** SetVolumeForListener **(auListener *listener, float dB)**

SetVolumeForListener - sets volume of this listener's voice to the specified auListener.

2. Public Attributes

- **GZ_DECLARE_TYPE_INTERFACE**

3. Detailed Description

Class: auListener

Function: Class to handle all aspects of ausim3D listener management. Contains methods to get/set positions, get/set voice radiation patterns, get/set HRTFs for individual listeners, and set

the offset for the listener's mouth. Permits output to **auServerGUI** browser windows, consoles, and text files.

4. Constructor and Destructor Documentation

- **auListener::auListener (const char * *name* = 0)**
    - Constructor.
    - Function: Constructor
    - Purpose: Create new listener
    - **Parameters:**
        - ***id*** - source id for this listener's voice
        - ***name*** - name of the listener

- **auListener::~auListener () `[virtual]`**
    - Destructor.
    - Function: Destructor
    - Purpose: Destroys the listener

5. Member Function Documentation

- **bool auListener::DecreaseMouthVolume (float *dBamount*)**
    - DecreaseMouthVolume - decreases the mouth volume by the specified factor.
    - Function: DecreaseMouthVolume
    - Purpose: Decreases the volume of the mouth - uniformly applied across radiation pattern if one exists
    - **Parameters:**
        - ***dBamount*** The amount of decibels to reduce the mouth volume Remember: a reduction of 3 dB reduces intensity by 50%

- **const char* auListener::GetHRTF () const `[inline]`**
    - GetHRTF - gets name of HRTF file

- **int auListener::GetID () const `[inline]`**
    - GetID - returns listener ID number

- **void auListener::GetMouthRadPattern (auRadPattern * *pattern*)**
    - GetMouthRadPattern - gets radiation pattern of mouth.
    - Function: GetMouthRadPattern
    - Purpose: Gets the radiation pattern of the mouth
    - **Parameters:**
        - ***pos*** - **auRadPattern** representing the radiation pattern of the mouth

- **void auListener::GetPosition (auPosition * *pos*)**
    - GetPosition - returns current position.
    - Function: GetPosition
    - Purpose: Gets the position of the listener
    - **Parameters:**

- ***pos*** **- auPosition** representing the location of the listener

- **bool auListener::IncreaseMouthVolume (float *dBamount*)**
  - IncreaseMouthVolume - increases the mouth volume by the specified factor.
  - Function: IncreaseMouthVolume
  - Purpose: Increases the volume of the mouth - uniformly applied across radiation pattern if one exists
  - **Parameters:**
    - ***dBamount*** - the amount of decibels to increase the mouth volume Remember: a increase of 3 dB increases intensity by 50%

- **bool auListener::RemExclusive ()**
  - RemExclusive - removes the exclusivity of this listener to the specified listener.

- **bool auListener::SetExclusive (auListener * *listener*)**
  - SetExclusive - sets this listener's voice exclusive to specified listener.

- **bool auListener::SetHRTF (const char * *hrtfName*)**
  - SetHRTF - sets user specified HRTF in ausim3D.
  - Function: SetHRTF
  - Purpose: Sets the HRTF of the listener
  - **Parameters:**
    - ***name*** - string for HRTF of listener

- **bool auListener::SetMouthOffset (auPosition * *pos*)**
  - SetMouthOffset - sets offset for mouth position from ears.
  - Function: SetMouthOffset
  - Purpose: Sets the position of the mouth relative to the ears
  - **Parameters:**
    - ***pos*** - **auPosition** representing the relative location of the mouth

- **bool auListener::SetMouthRadPattern (auRadPattern * *pattern*)**
  - SetMouthRadPattern - sets radiation pattern of mouth.
  - Function: SetMouthRadPattern
  - Purpose: Sets the radiation pattern of the mouth
  - **Parameters:**
    - ***pos*** - **auRadPattern** representing the radiation pattern of the mouth

- **bool auListener::SetMouthVolume (float *dB*)**
  - SetMouthVolume - sets the overall volume of the mouth source.
  - Function: SetMouthVolume
  - Purpose: Sets the volume of the mouth - uniformly applied across radiation pattern if one exists
  - **Parameters:**

- ***dB*** The volume, in decibels; vol < -120 is equivalent to turning off source Remember: a reduction of 3 dB reduces intensity by 50%

- **bool auListener::SetPosition (auPosition * *pos* = NULL)**
  - SetPosition - updates position.
  - Function: SetPosition
  - Purpose: Sets the position of the listener
  - **Parameters:**
    - ***posit*** - **auPosition** for the location of the listener

- **bool auListener::SetVoiceInputChannel (int *channel*)**
  - SetVoiceInputChannel - sets the channel for voice input.
  - Function: SetVoiceInputChannel
  - Purpose: Sets the voice input channel on ausim3D
  - **Parameters:**
    - ***channel*** - channel number for voice input

- **bool auListener::SetVolumeForListener (auListener * *otherListener*, float *dB*)**
  - SetVolumeForListener - sets volume of this listener's voice to the specified auListener.
  - Function: SetVolumeForListener
  - Purpose: sets this channel's volume for a specified listener
  - **Parameters:**
    - ***listener*** - the auListener
    - ***dB*** volume of the source in dB; 0.0f is maximum (original source volume) and negative values reduce volume intensity Remember: a reduction of 3 dB reduces intensity by 50%

6. Member Data Documentation

- **auListener::GZ_DECLARE_TYPE_INTERFACE**

  - Reimplemented from **auSource**.

The documentation for this class was generated from the following files:

- **aulistener.h**
- **aulistener.cpp**

E. AUPOSITION CLASS REFERENCE

```
#include <auposition.h>
```

1. Public Methods

- **auPosition ()**

Constructor - default.

- auPosition **(const float posit[])**

Constructor - takes array of 6 floats.

- auPosition **(const float x, const float y, const float z, const float h, const float p, const float r)**

Constructor - takes 6 individual floats.

- **virtual** ~auPosition **()**

Destructor.

- **void** Set **(const float x, const float y, const float z, const float h, const float p, const float r)**

Set - sets position from 6 individual floats.

- **float** X **(void)**
- **float** Y **(void)**
- **float** Z **(void)**
- **float** H **(void)**
- **float** P **(void)**
- **float** R **(void)**
- **void** X **(float x)**

X - sets x value.

- **void** Y **(float y)**

Y - sets y value.

- **void** Z **(float z)**

Z - sets z value.

- **void** H **(float h)**

H - sets heading.

- **void** P **(float p)**

P - sets pitch.

- **void** R **(float r)**

R - sets roll.

2. Public Attributes

- **GZ_DECLARE_TYPE_INTERFACE**

3. Detailed Description

Class: auPosition

Function: Simple class to represent six floats for source/listener position. Six floats represent x, y, z positions and heading, pitch, and roll. Ausim3D uses a right-handed coordinate system where the default is x-forward, y-left, and z-up.

4. Constructor and Destructor Documentation

- **auPosition::auPosition ()**
    - Constructor - default.
    - Function: Constructor
    - Purpose: Create new auPosition object with all values - 0.0f

- **auPosition::auPosition (const float *posit*[])**
    - Constructor - takes array of 6 floats.
    - Function: Constructor
    - Purpose: Create new auPosition object
    - **Parameters:**
        - ***posit*** - array of 6 floats for x,y,z,h,p,r

- **auPosition::auPosition (const float *x*, const float *y*, const float *z*, const float *h*, const float *p*, const float *r*)**
    - Constructor - takes 6 individual floats.
    - Function: Constructor
    - Purpose: Create new auPosition object
    - **Parameters:**
        - ***x*** - the x value
        - ***y*** - the y value
        - ***z*** - the z value
        - ***h*** - the h value
        - ***p*** - the p value
        - ***r*** - the r value

- **auPosition::~auPosition () `[virtual]`**
    - Destructor.
    - Function: Destructor
    - Purpose: Destroy this auPosition object

5. Member Function Documentation

- **void auPosition::H (float *h*)**
    - H - sets heading.
    - Function: H
    - Purpose: Input H into auPosition
    - **Parameters:**
        - ***h*** - the new heading

- **float auPosition::H (void) `[inline]`**
    - H - gets heading

- **void auPosition::P (float *p*)**
    - P - sets pitch.
    - Function: P
    - Purpose: Input P into auPosition
    - **Parameters:**
        - ***px*** - the new pitch

- **float auPosition::P (void) `[inline]`**
    - P - gets pitch

- **void auPosition::R (float *r*)**
    - R - sets roll.
    - Function: R
    - Purpose: Input R into auPosition
    - **Parameters:**
        - ***r*** - the new roll

- **float auPosition::R (void) `[inline]`**
    - R - gets roll

- **void auPosition::Set (const float *x*, const float *y*, const float *z*, const float *h*, const float *p*, const float *r*)**
    - Set - sets position from 6 individual floats.
    - Function: Set
    - Purpose: Set the values of the auPosition
    - **Parameters:**
        - ***x*** - the x value
        - ***y*** - the y value
        - ***z*** - the z value
        - ***h*** - the h value
        - ***p*** - the p value
        - ***r*** - the r value

- **void auPosition::X (float *x*)**
    - X - sets x value.
    - Function: X
    - Purpose: Input X into auPosition
    - **Parameters:**
        - ***x*** - the new x position

- **float auPosition::X (void) `[inline]`**
    - X - gets x value

- **void auPosition::Y (float *y*)**
    - Y - sets y value.
    - Function: Y
    - Purpose: Input Y into auPosition
    - **Parameters:**
        - ***y*** - the new y position

- **float auPosition::Y (void) `[inline]`**
    - Y - gets y value

- **void auPosition::Z (float *z*)**
    - Z - sets z value.
    - Function: Z
    - Purpose: Input Z into auPosition
    - **Parameters:**
        - ***z*** - the new z position

- **float auPosition::Z (void) `[inline]`**
    - Z - gets z value

6. Member Data Documentation

- **auPosition::GZ_DECLARE_TYPE_INTERFACE**
    - Reimplemented from **auRefData**.

The documentation for this class was generated from the following files:

- **auposition.h**
- **auposition.cpp**

F. AURADPATTERN CLASS REFERENCE

`#include <auradpattern.h>`

1. Public Methods

- auRadPattern **(float deg0=0.0f, float deg90=0.0f, float deg180=0.0f)**

Constructor.

- **virtual ~auRadPattern ()**

Destructor.

- **float** Get **(int index)**

Get - gets value at specified index.

- **void** Set **(int index, float value)**

Set - sets the value at the specified index.

- **int** GetSize **()**

2. Public Attributes

- **GZ_DECLARE_TYPE_INTERFACE**

3. Detailed Description

Class: auRadPattern

Function: Simple class to represent three floats for source/voice radiation patterns. Three floats represent radiation pattern at 0 deg, +/- 90 deg, and 180 deg relative to source.

4. Constructor and Destructor Documentation

- **auRadPattern::auRadPattern (float *deg0* = 0.0f, float *deg90* = 0.0f, float *deg180* = 0.0f)**
    - Constructor.
    - Function: Constructor
    - Purpose: creates new auRadPattern with all values = 0.0f

- **auRadPattern::~auRadPattern () `[virtual]`**
    - Destructor.
    - Function: Destructor
    - Purpose: Destroys this auRadPattern

5. Member Function Documentation

- **float auRadPattern::Get (int *index*)**
    - Get - gets value at specified index.
    - Function: Get
    - Purpose: Get the value at the specified index
    - **Parameters:**
        - ***index*** - the index
    - **Returns:**
        - float - the value of the radiation pattern at the specified index if index out of bounds, returns -9999.0

- **int auRadPattern::GetSize () `[inline]`**
    - GetSize - returns number of elements in radiation pattern

- **void auRadPattern::Set (int *index*, float *value*)**
    - Set - sets the value at the specified index.
    - Function: Set
    - Purpose: Set the value at the specified index
    - **Parameters:**
        - ***index*** - the index

- ***value*** - the value of the radiation pattern at the specified index

6. Member Data Documentation

- **auRadPattern::GZ_DECLARE_TYPE_INTERFACE**
  - Reimplemented from **auRefData**.

The documentation for this class was generated from the following files:

- **auradpattern.h**
- **auradpattern.cpp**

G. AUREFDATA CLASS REFERENCE

`#include <aurefdata.h>`

1. Public Methods

- **auRefData ()**
- **virtual** ~auRefData **()**

2. Public Attributes

- **GZ_DECLARE_TYPE_INTERFACE**

3. Detailed Description

Class: auRefData
Function: Class to package data packets for transmission within AuServerLib

4. Constructor and Destructor Documentation

- **auRefData::auRefData ()**
  - Function: Constructor
  - Purpose: Create new auRefData

- **auRefData::~auRefData () `[virtual]`**
  - Function: Destructor
  - Purpose: Destroys auRefData

5. Member Data Documentation

- **auRefData::GZ_DECLARE_TYPE_INTERFACE**
  - Reimplemented in **auPosition**, and **auRadPattern**.

The documentation for this class was generated from the following files:

- **aurefdata.h**
- **aurefdata.cpp**

H. AUSERVERGUI CLASS REFERENCE

`#include <auserverGUI.h>`

1. Public Methods

- **auServerGUI ()**

Constructor.

- **virtual** ~auServerGUI **()**

Destructor.

2. Detailed Description

Class: auServerGUI

Function: Class for displaying pertinent information about auListeners, auSounds, and auChannels. Contains methods for manipulating all parameters of these objects. Contains methods for setting global parameters: Rolloff and Absorption.

3. Constructor and Destructor Documentation

- **auServerGUI::auServerGUI ()**
    - Constructor.
    - Function: Constructor
    - Purpose: creates new auServerGUI

- **auServerGUI::~auServerGUI () `[virtual]`**
    - Destructor.
    - Function: Destructor
    - Purpose: Destroys the gui

The documentation for this class was generated from the following files:

- **auserverGUI.h**
- **auserverGUI.cpp**

I. AUSOUND CLASS REFERENCE

`#include <ausound.h>`

1. Public Methods

- auSound **(const char *filename, const char *name)**

Constructor.

- **virtual** ~auSound **()**

Destructor.

- **bool** Play **()**

Play - plays sound.

- **bool** IsPlaying **()**

IsPlaying - tells whether sound is playing; TRUE = playing.

- **bool** Loop **()**

Loop - loops sound.

- **bool** Stop **()**

Stop = stops playing.

- **bool** Rewind **()**

Rewind - rewinds sound to beginning.

- **bool LinkToListener (auListener *listener)**

*LinkToListener - links source to specified **auListener**.*

- **bool** UnLink **()**

Unlink - unlinks source to all auListeners.

- **bool SetExclusive (auListener *listener)**

SetExclusive - directs this channel exclusively to specified listener.

- **bool RemExclusive ()**

RemExclusive - removes any exclusivity of this source to any listener.

- **bool SetVolumeForListener (auListener *listener, float dB)**

SetVolumeForListener - sets volume to the specified listener.

2. Public Attributes

- **GZ_DECLARE_TYPE_INTERFACE**

3. Detailed Description

Class: auSound

Function: Class representing all aspects of an ausim source generated from a wave file. Contains methods for setting/getting source position, radiation patterns, names, and source-specific rolloff. Contains methods to link/unlink to specified listener.

4. Constructor and Destructor Documentation

- **auSound::auSound (const char * *filename*, const char * *name*)**
    - Constructor.
    - Function: Constructor
    - Purpose: Creates new auSound object
    - **Parameters:**
        - ***source*** - source ID for this sound
        - ***name*** - name of this sound - also the name of the wav file

- **auSound::~auSound () `[virtual]`**
    - Destructor.
    - Function: Destructor
    - Purpose: Destroys this source; closes wav file structure

5. Member Function Documentation

- **bool auSound::IsPlaying ()**
    - IsPlaying - tells whether sound is playing; TRUE = playing.
    - Function: IsPlaying
    - Purpose: Indicated whether sound is playing; TRUE = playing
    - **Returns:**
        - bool - true = playing; false = not playing

- **bool auSound::LinkToListener (auListener * *listener*)**
    - LinkToListener - links source to specified **auListener**.
    - Function: LinkToListener
    - Purpose: Link this source to the specified **auListener**
    - **Parameters:**
        - ***listener*** - the **auListener** to link with

- **bool auSound::Loop ()**
    - Loop - loops sound.
    - Function: Loop
    - Purpose: Loops this sound continuously until **Stop**() called

- **bool auSound::Play ()**
    - Play - plays sound.
    - Function: Play
    - Purpose: Plays this sound once

- **bool auSound::RemExclusive ()**
    - RemExclusive - removes any exclusivity of this source to any listener.
    - Function: RemExclusive
    - Purpose: removes any exclusivity setting from this sound

- **bool auSound::Rewind ()**

- Rewind - rewinds sound to beginning.
- Function: Rewind
- Purpose: Rewinds the play counter for the wav file to the starting position

- **bool auSound::SetExclusive (auListener * *listener*)**
  - SetExclusive - directs this channel exclusively to specified listener.
  - Function: SetExclusive
  - Purpose: sets this sound to only be heard by specified listener
  - **Parameters:**
    - ***listener*** - the exclusive **auListener** to be heard by

- **bool auSound::SetVolumeForListener (auListener * *listener*, float *volume*)**
  - SetVolumeForListener - sets volume to the specified listener.
  - Function: SetVolumeForListener
  - Purpose: sets this channel's volume for a specified listener
  - **Parameters:**
    - ***listener*** - the **auListener**

- **bool auSound::Stop ()**
  - Stop = stops playing.
  - Function: Stop
  - Purpose: Stops play on this sound

- **bool auSound::UnLink ()**
  - Unlink - unlinks source to all auListeners.
  - Function: LinkToListener
  - Purpose: Link this source to the specified **auListener**
  - **Parameters:**
    - ***listener*** - the **auListener** to link with

6. Member Data Documentation

- **auSound::GZ_DECLARE_TYPE_INTERFACE**
  - Reimplemented from **auSource**.

The documentation for this class was generated from the following files:

- **ausound.h**
- **ausound.cpp**

J. AUSOURCE CLASS REFERENCE

```
#include <ausource.h>
```

1. Public Methods

- **auSource ()**

Constructor.

- **virtual** ~auSource **()**

Destructor.

- **bool SetPosition (auPosition *pos)**

SetPosition - sets source's position.

- **void GetPosition (auPosition *pos)**

GetPosition - gets source's position.

- **bool SetRadPattern (auRadPattern *pattern)**

SetRadPattern - sets radiation pattern.

- **void GetRadPattern (auRadPattern *pattern)**

GetRadPattern - gets source's radiation pattern.

- **bool** SetRolloff **(float factor)**

SetRolloff - sets source's rolloff.

- **const float** GetRolloff **()**
- **const char *** GetLinkName **()**
- **bool** SetSpatial **(bool spatial=true)**

SetSpatialOff - removes source from spatialization.

- **bool** SetPan **(float panLeftDB=0.0f, float panRightDB=0.0f)**

SetPan - sets the left and right pan settings.

- **float** GetPanLeft **()**

GetPanLeft - returns pan left setting when spatialization off; 9999 indicates spatialization on.

- **float GetPanRight ()**

GetPanRight - returns pan right setting when spatialization off; 9999 indicates spatialization on.

- **int** GetID **()**
- **bool** SetVolume **(float dB)**

SetVolume - sets the volume of the source in dB.

2. Public Attributes

- **GZ_DECLARE_TYPE_INTERFACE**

3. Protected Methods

- **bool SetSpatialSettings ()**

SetSpatialSettings - sets the spatial and pan settings.

4. Protected Attributes

- **char** mName **[128]**
- **char** mLinkName **[128]**
- **int mSourceID**
- **auRadPattern * mRad**
- **auPosition * mPos**
- **float mRolloff**
- **float mPanLeft**
- **float mPanRight**
- **bool mSpatial**
- **float** mVolume
- **bool isConfigured**

5. Detailed Description

Class: auSource

Function: Base class for **auSound** and **auChannel** classes. Contains methods for setting/getting source position, radiation patterns, names, and source-specific rolloff. Contains methods to link/unlink to specified listener. NOTE: Developers should NOT directly instantiate objects of this class.

6. Constructor and Destructor Documentation

- **auSource::auSource ()**
    - Constructor.
    - Function: Constructor
    - Purpose: Creates new auSource object

- **auSource::~auSource () `[virtual]`**
    - Destructor.
    - Function: Destructor
    - Purpose: Destroy this source

7. Member Function Documentation

- **int auSource::GetID () `[inline]`**
    - GetID - returns source ID

- **const char* auSource::GetLinkName () `[inline]`**

- GetLinkName - returns name of **auListener** linked with

- **float auSource::GetPanLeft ()**
  - GetPanLeft - returns pan left setting when spatialization off; 9999 indicates spatialization on.
  - Function: GetPanLeft
  - Purpose: gets the pan left setting when spatialization is off; 9999 indicates spatialization on
  - **Returns:**
    - pan left setting, if not spatial return 9999.0

- **float auSource::GetPanRight ()**
  - GetPanRight - returns pan right setting when spatialization off; 9999 indicates spatialization on.
  - Function: GetPanRight
  - Purpose: gets the pan right setting when spatialization is off; 9999 indicates spatialization on
  - **Returns:**
    - pan right settings

- **void auSource::GetPosition (auPosition * *pos*)**
  - GetPosition - gets source's position.
  - Function: GetPosition
  - Purpose: Get the position of this source
  - **Parameters:**
    - **pos** - **auPosition** of source
  - Reimplemented in **auListener**.

- **void auSource::GetRadPattern (auRadPattern * *pattern*)**
  - GetRadPattern - gets source's radiation pattern.
  - Function: GetRadPattern
  - Purpose: Get the radiation pattern of this source
  - **Parameters:**
    - ***pattern*** - **auRadPattern** of source

- **const float auSource::GetRolloff () `[inline]`**
  - GetRolloff - gets source's rolloff factor

- **bool auSource::SetPan (float *panLeftDB* = 0.0f, float *panRightDB* = 0.0f)**
  - SetPan - sets the left and right pan settings.
  - Function: SetPan
  - Purpose: sets this source as spatialized with specified pan settings
  - **Parameters:**
    - ***panLeftDB*** gain for left output when non-spatialized; 0.0 = full volume, -120.0 = fully attenuated

        - ***panRightDB*** gain for right output when non-spatialized; 0.0 = full volume, -120.0 = fully attenuated

- **bool auSource::SetPosition (auPosition * *pos*)**
    - SetPosition - sets source's position.
    - Function: SetPosition
    - Purpose: Set the position of this source
    - **Parameters:**
        - **pos** - **auPosition** of source
    - Reimplemented in **auListener**.

- **bool auSource::SetRadPattern (auRadPattern * *pattern*)**
    - SetRadPattern - sets radiation pattern.
    - Function: SetRadPattern
    - Purpose: Set the radiation pattern of this source
    - **Parameters:**
        - ***pattern*** - **auRadPattern** of source

- **bool auSource::SetRolloff (float *factor*)**
    - SetRolloff - sets source's rolloff.
    - Function: SetRolloff
    - Purpose: Sets the source specific rolloff multiplier - multiplies against global rolloff factor range limited to (0.0f, 5.0f) 0.0f = no individualized source-specific rolloff factor 5.0f = maximum source-specific rolloff multiplicative factor
    - **Parameters:**
        - ***factor*** - multiplicative factor for source specific rolloff

- **bool auSource::SetSpatial (bool *spatial* = true)**
    - SetSpatialOff - removes source from spatialization.
    - Function: SetSpatial
    - Purpose: sets this source as either non-spatialized or spatialized
    - **Parameters:**
        - ***spatial*** - bool indicating whether spatial or not

- **bool auSource::SetSpatialSettings () `[protected]`**
    - SetSpatialSettings - sets the spatial and pan settings.
    - Function: SetSpatialSettings
    - Purpose: sets this source's spatialization and pan settings

- **bool auSource::SetVolume (float *dB*)**
    - SetVolume - sets the volume of the source in dB.
    - Function: SetVolume
    - Purpose: sets the volume of this source
    - **Parameters:**

- ***dB*** volume of the source in dB; 0.0f is maximum (original source volume) and negative values reduce volume intensity Remember: a reduction of 3 dB reduces intensity by 50%.

8. Member Data Documentation

- **auSource::GZ_DECLARE_TYPE_INTERFACE**
  - Reimplemented from **auBase**.

- **bool auSource::isConfigured `[protected]`**
  - indicates whether source is configured

- **char auSource::mLinkName[128] `[protected]`**
  - name of the listener this source is linked to

- **char auSource::mName[128] `[protected]`**
  - name of this source

- **float auSource::mPanLeft `[protected]`**
  - pan left setting for this source when spatialization off

- **float auSource::mPanRight `[protected]`**
  - pan right setting for this source when spatialization off

- **auPosition* auSource::mPos `[protected]`**
  - the position of this source

- **auRadPattern* auSource::mRad `[protected]`**
  - radiation pattern for this source

- **float auSource::mRolloff `[protected]`**
  - rolloff factor of this source

- **int auSource::mSourceID `[protected]`**
  - source ID number

- **bool auSource::mSpatial `[protected]`**
  - indicates whether spatialization is ON (true) or OFF (false)

- **float auSource::mVolume `[protected]`**
  - stored volume setting for this source

The documentation for this class was generated from the following files:

- **ausource.h**
- **ausource.cpp**

K. AUSYSTEM CLASS REFERENCE

`#include <ausystem.h>`

1. Public Methods

- auSystem **(const char *name=0)**

Constructor.

- **virtual** ~auSystem **()**

Destructor.

- **void** Run **(void)**

Set the system to Run. This is a blocking call.

- **bool** IsConfigured **(void) const**
- **void** Config **(void)**

Configure the system.

- **void** Exit **()**

Exit - quit application.

- **void** Init **(int argc=0, char **argv=NULL)**

Init - init the application.

- **void** SetUpdateInterval **(const double interval)**

SetUpdateInterval - Set the tick interval time.

2. Public Attributes

- **GZ_DECLARE_TYPE_INTERFACE**

3. Constructor and Destructor Documentation

- **auSystem::auSystem (const char * *name* = 0)**
  - Constructor.
  - Function: Constructor
  - Purpose: Creates new auSysytem

- **auSystem::~auSystem () `[virtual]`**
  - Destructor.

- Function: Destructor
- Purpose: destroys auSysytem

4. Member Function Documentation

- **void auSystem::Config (void)**
  - Configure the system.
  - Function: Config
  - Purpose: Public function to configure the system. Typically called once to finish setting up all the au classes.

- **void auSystem::Exit ()**
  - Exit - quit application.
  - Function: Exit
  - Purpose: exits from auSysytem and shuts down

- **void auSystem::Init (int *argc* = 0, char ** *argv* = NULL)**
  - Init - init the application.

- **bool auSystem::IsConfigured (void) const `[inline]`**
  - Has the auSystem been configured yet?

- **void auSystem::Run (void)**
  - Set the system to Run. This is a blocking call.
  - Function: Run
  - Purpose: Sets the auSystem in motion, never to return until an exit event is triggered.

- **void auSystem::SetUpdateInterval (const double *interval*)**
  - Set the tick interval time.
  - Function: SetTickInterval
  - Purpose: Set the time interval that the system should use for the "tick" message. This is an optional message that gets reliably sent out every interval seconds.
  - **Parameters:**
    - ***interval*** - In seconds

5. Member Data Documentation

- **auSystem::GZ_DECLARE_TYPE_INTERFACE**
  - Reimplemented from **auBase**.

The documentation for this class was generated from the following files:

- **ausystem.h**
- **ausystem.cpp**

L. UPDATEGUI CLASS REFERENCE

`#include <auserverGUI.h>`

1. Public Methods

- **UpdateGUI ()**

Constructor.

- **virtual** ~UpdateGUI **()**

Destructor.

2. Detailed Description

Class: UpdateGUI

Function: Simple thread class to manage continuous updating of gui parameters.

NOTE: Developers should NOT directly instantiate objects of this class. Use of **auServerGUI** will automatically create this thread for automatic updating.

3. Constructor and Destructor Documentation

- **UpdateGUI::UpdateGUI ()**
    - Constructor.
    - Function: Constructor
    - Purpose: creates new UpdateGUI thread

- **UpdateGUI::~UpdateGUI () `[virtual]`**
    - Destructor.
    - Function: Destructor
    - Purpose: Destroys UpdateGUI thread

The documentation for this class was generated from the following files:

- **auserverGUI.h**
- **auserverGUI.cpp**

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. VOICE LATENCY DATA

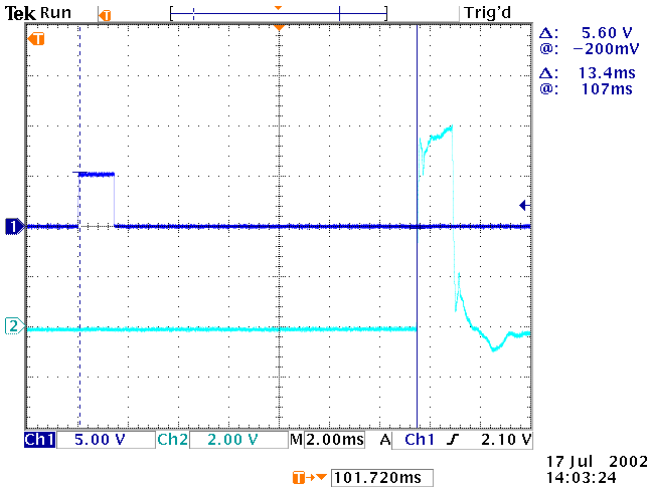
A. INTRODUCTION

The following table lists the observed latency measurements for both the Ausim3D GoldServe and DirectVoice VoIP live voice implementations.

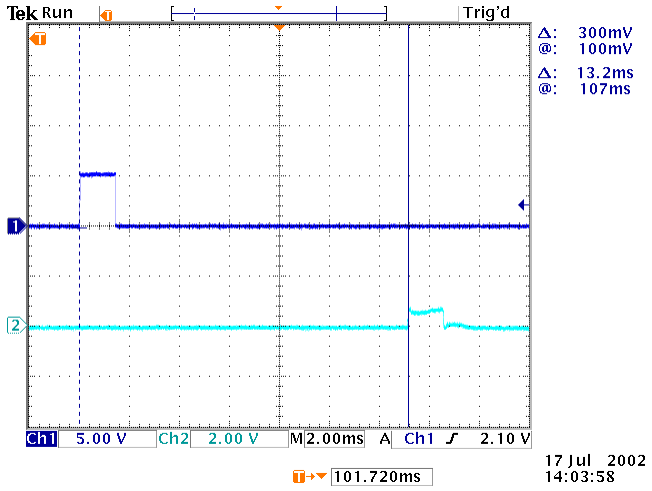
Run	GoldServe	DirectVoice
1	13.4	186
2	13.2	212
3	13.2	234
4	13.1	196
5	13.0	206
6	13.4	214
7	13.5	202
8	12.8	218
9	13.1	186
10	13.0	220
11	13.4	190
12	13.4	230
13	13.1	196
14	13.1	226
15	12.7	192
16	13.3	222
17	13.0	184
18	12.9	216
19	13.0	190
20	13.2	218
21	13	190
22	13	216
23	12.8	192
24	--	222
25	--	194
26	--	186
27	--	204
28	--	186
29	--	218
30	--	200

B. AUSIM3D GOLDSERVE DATA

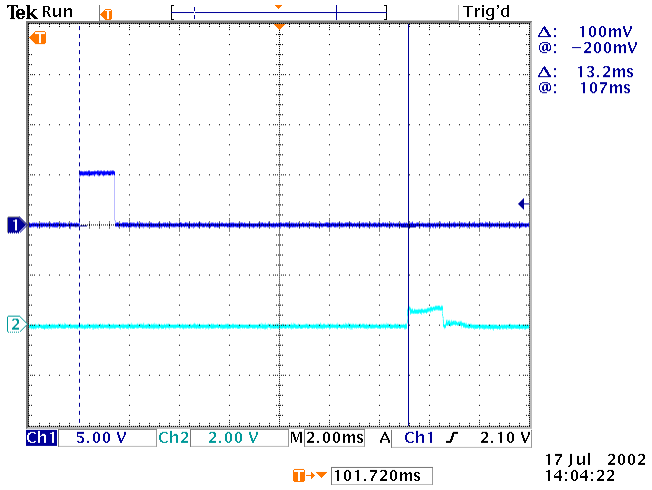
Run 1:



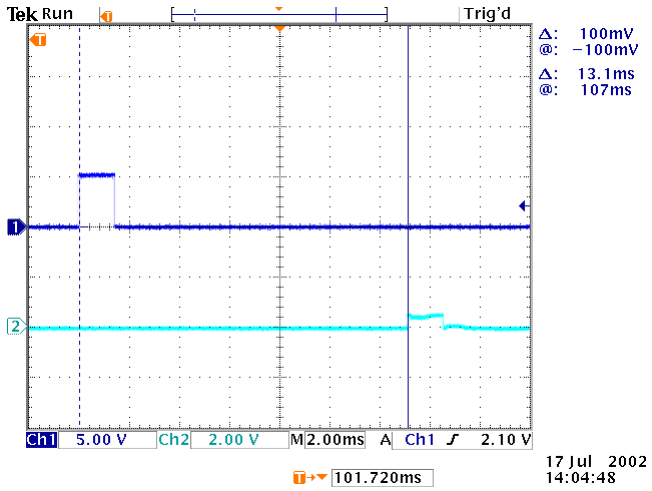
Run 2:



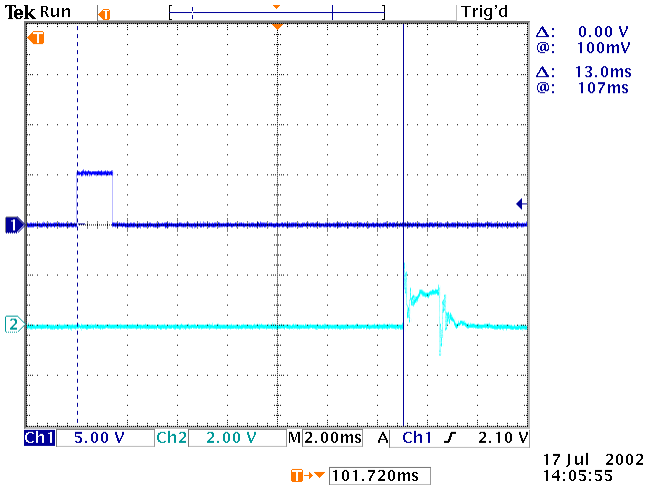
Run 3:



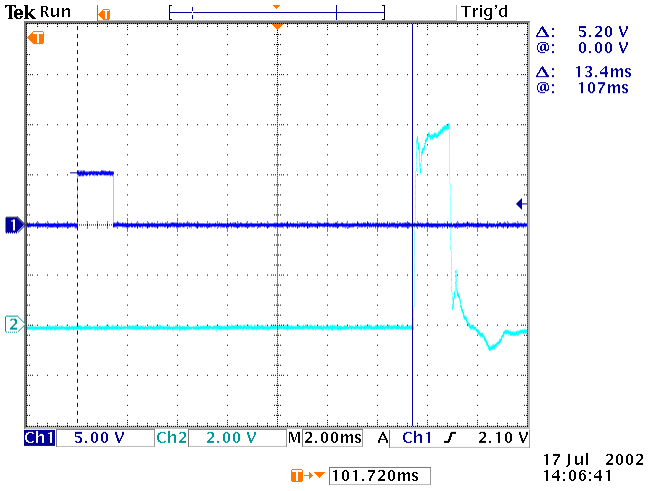
Run 4:



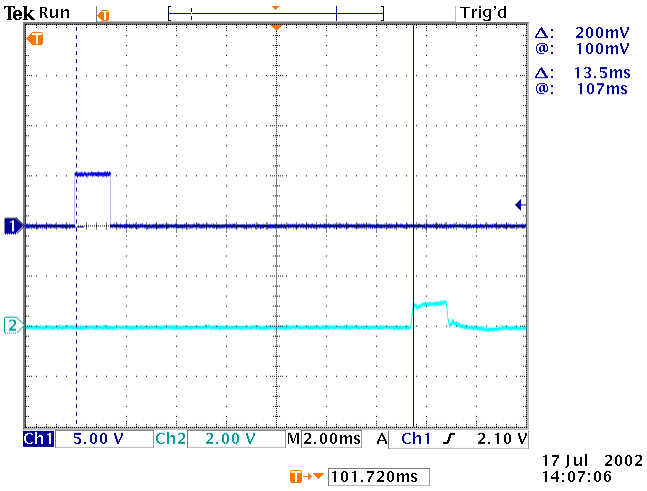
Run 5:



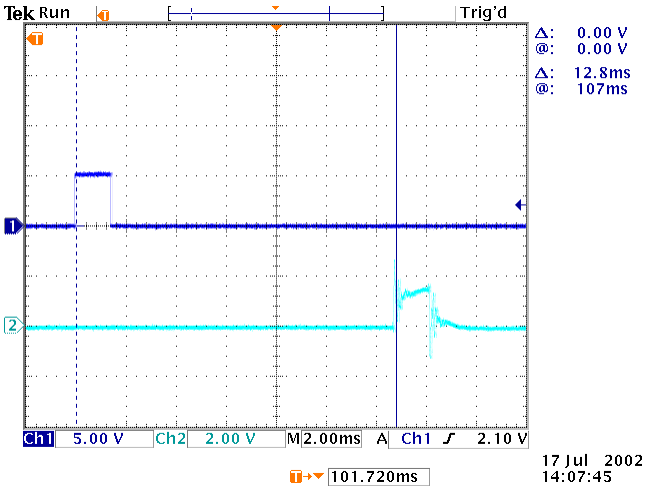
Run 6:



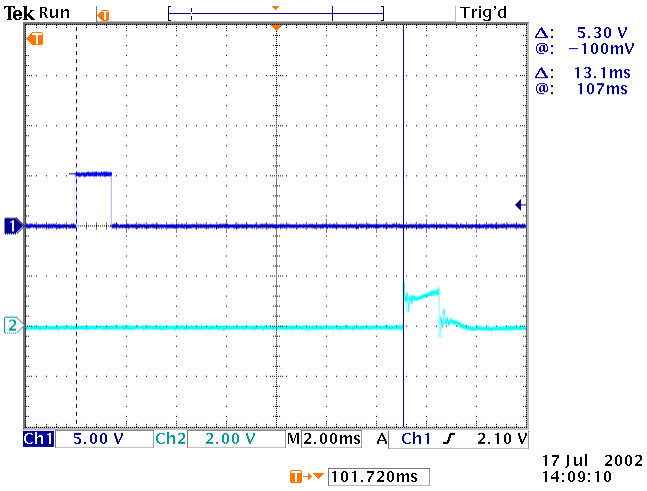
Run 7:



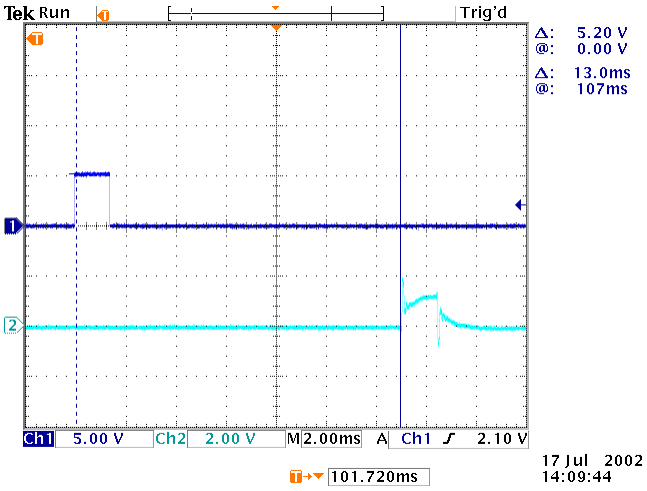
Run 8:



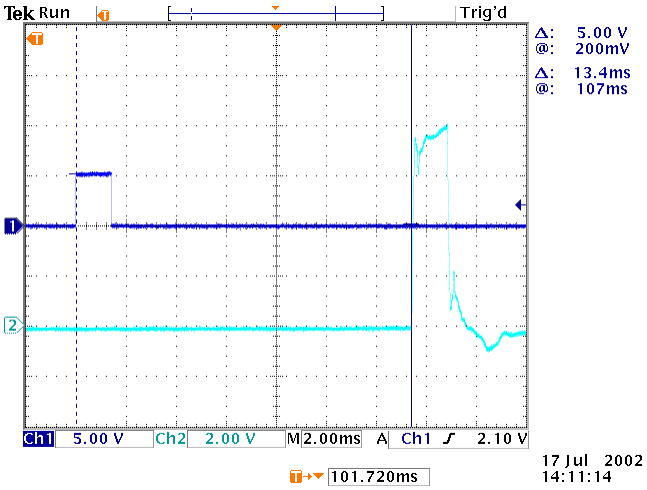
Run 9:



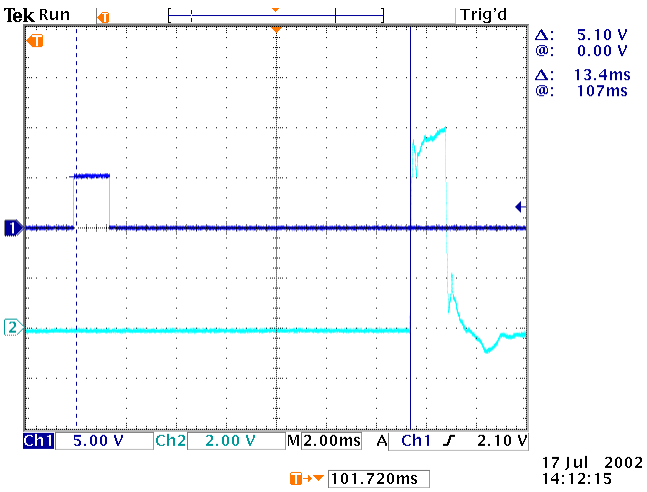
Run 10:



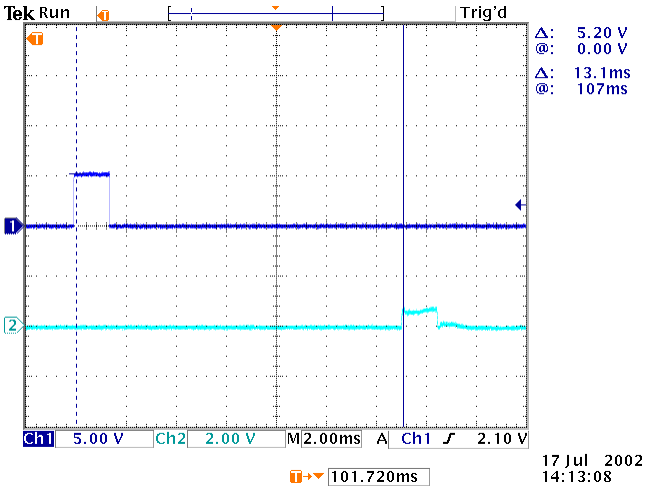
Run 11:



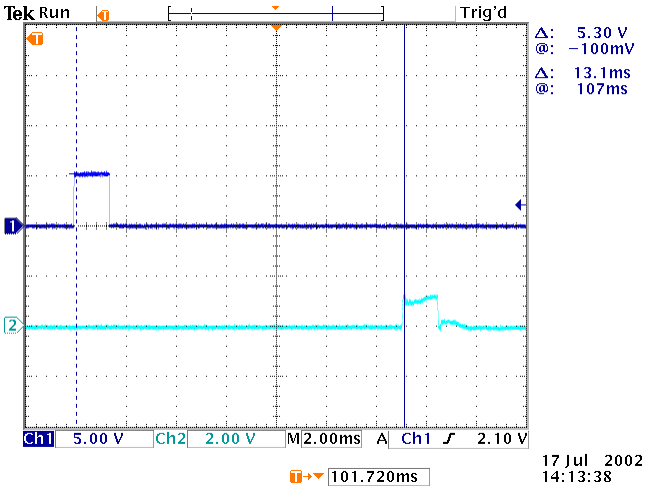
Run 12:



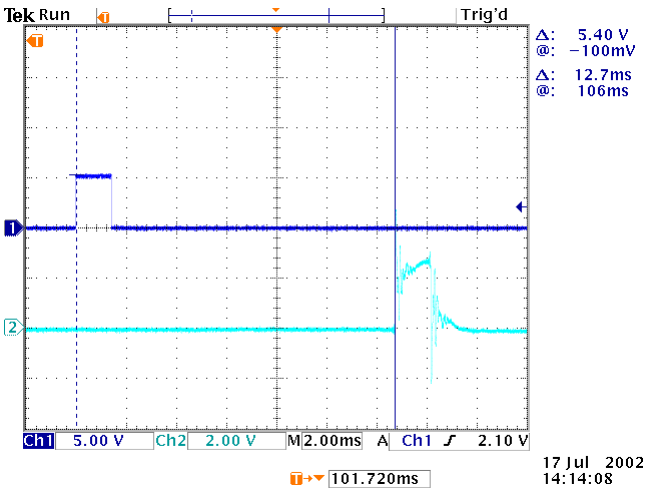
Run 13:



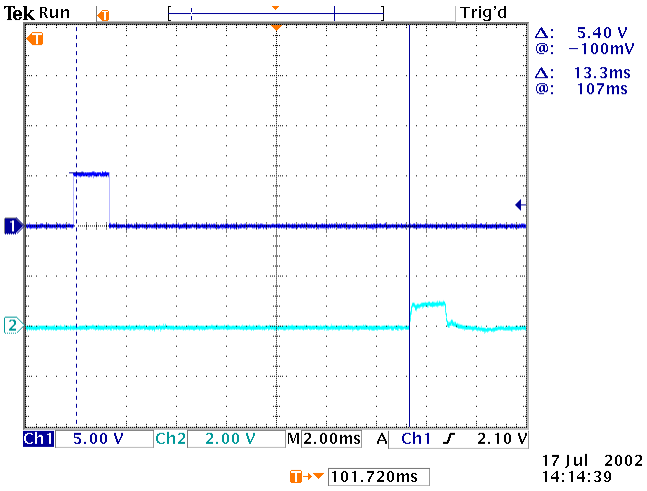
Run 14:



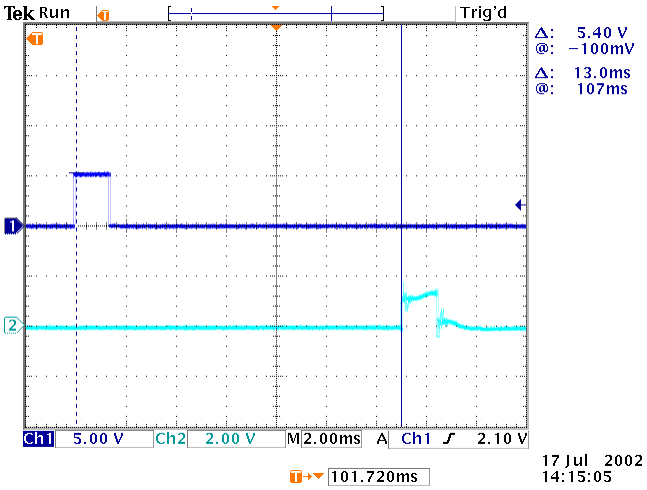
Run 15:



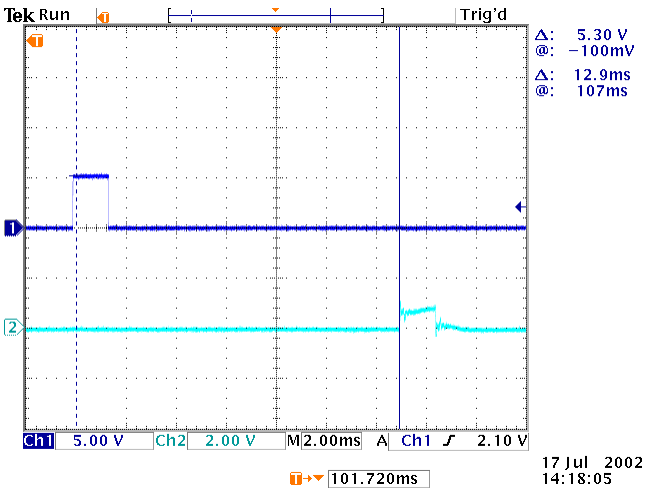
Run 16:



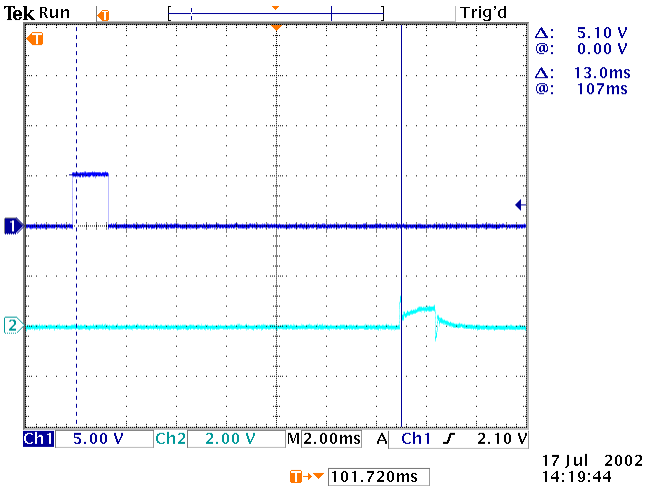
Run 17:



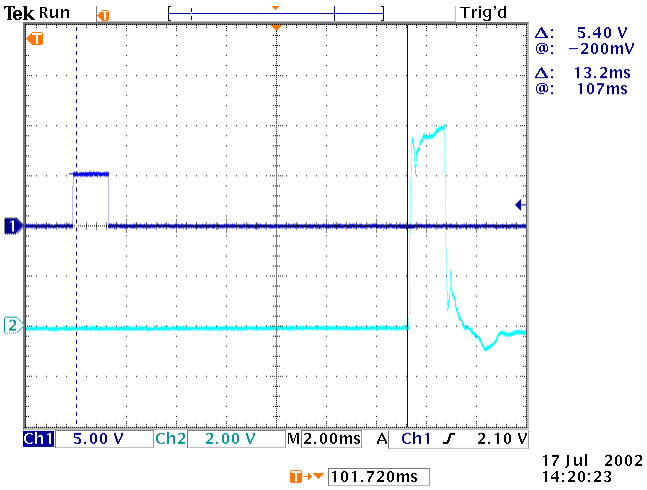
Run 18:



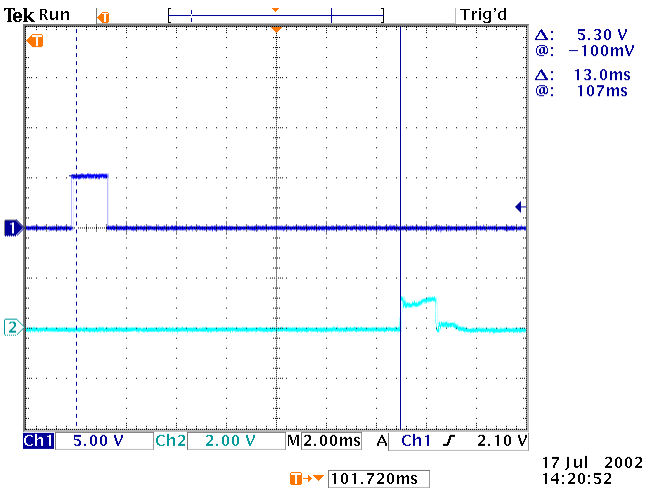
Run 19:



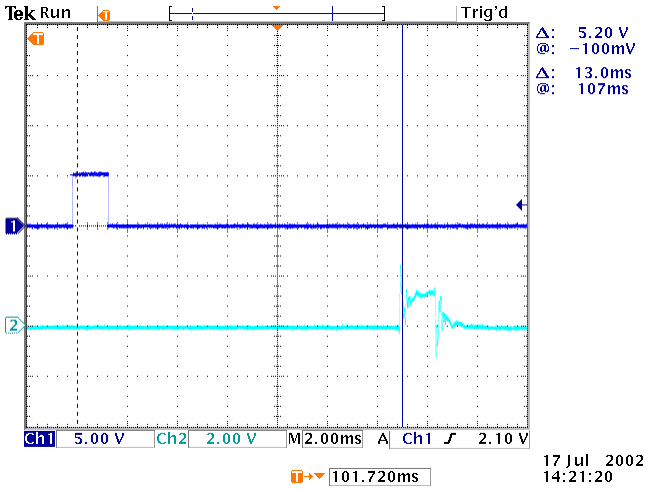
Run 20:



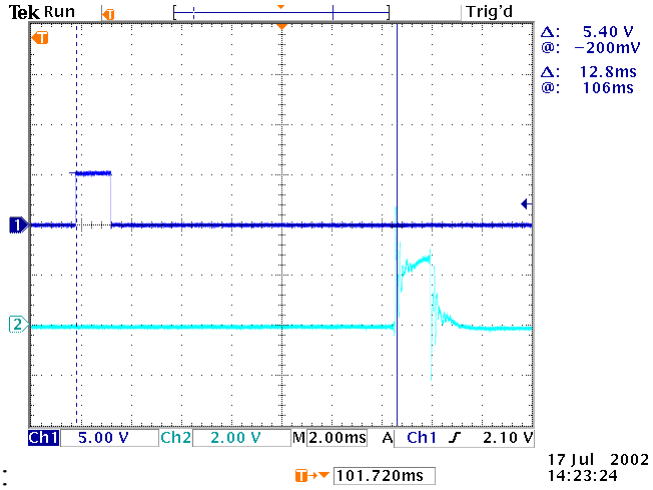
Run 21:



Run 22:



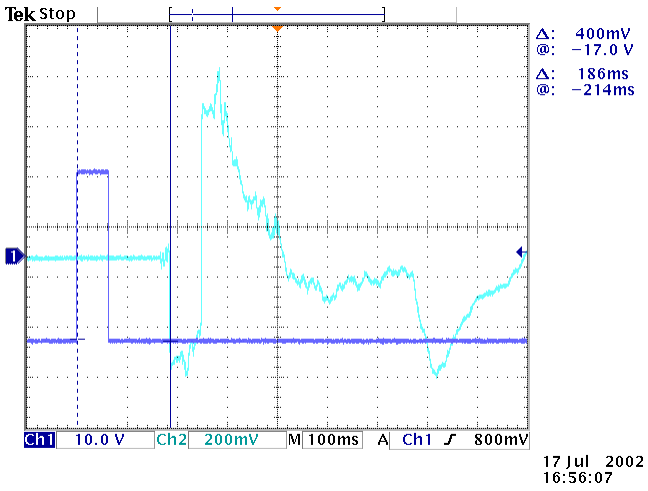
Run 23:



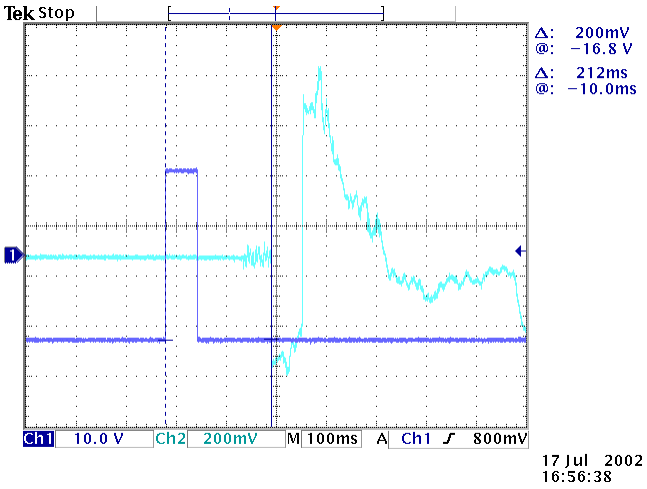
:

C. DIRECTVOICE DATA

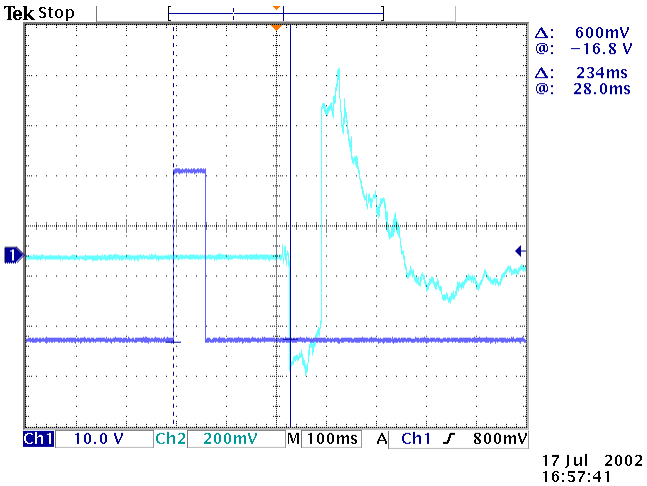
Run 1:



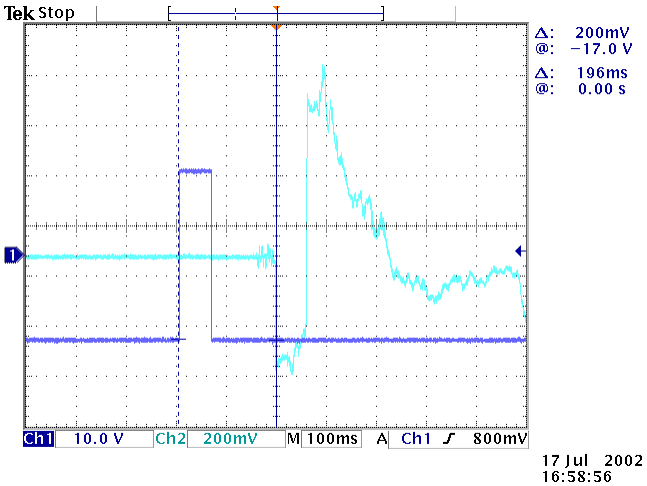
Run 2:



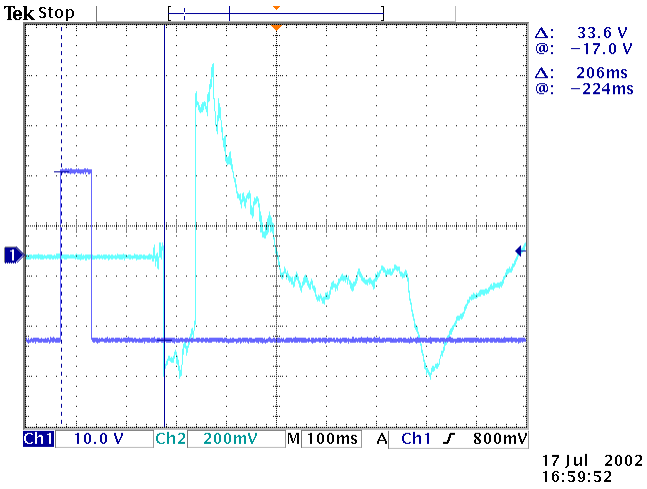
Run 3:



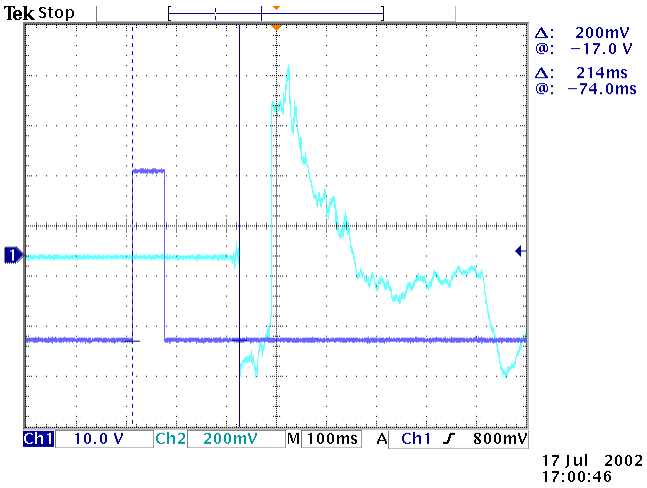
Run 4:



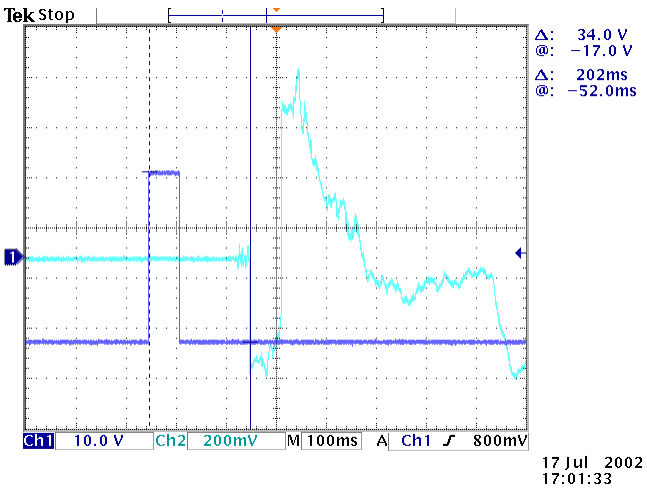
Run 5:



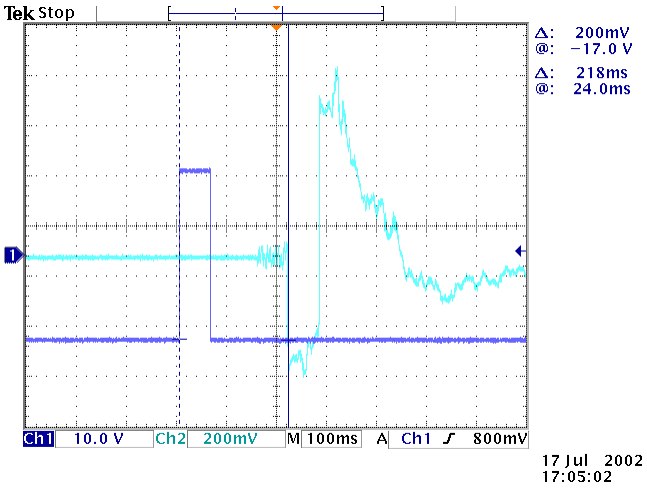
Run 6:



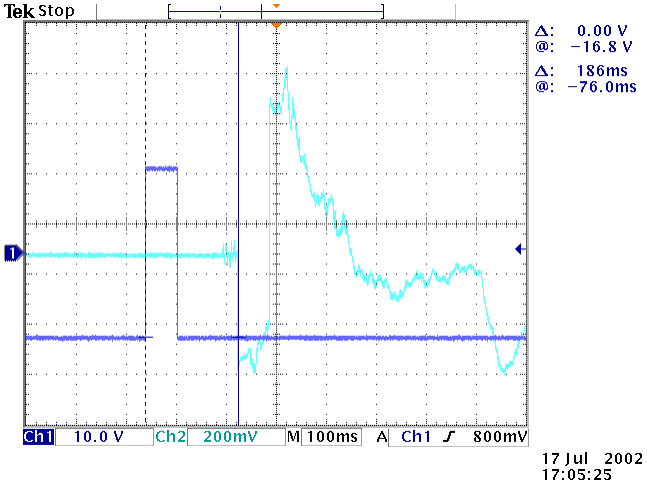
Run 7:



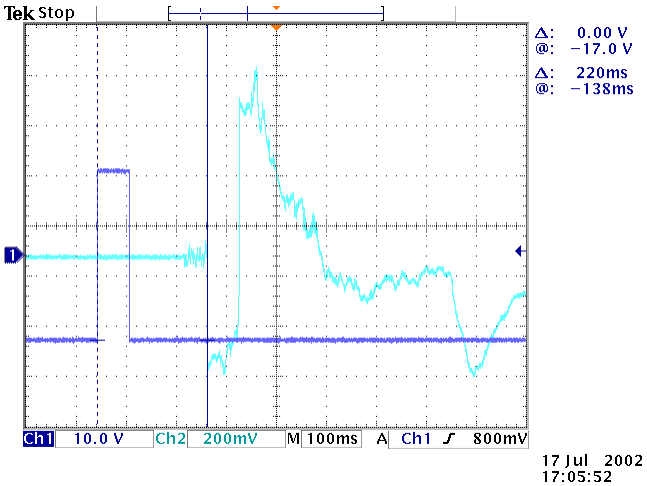
Run 8:



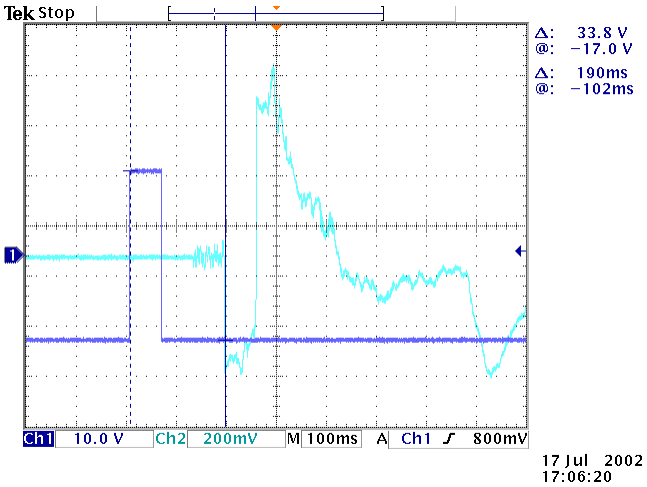
Run 9:



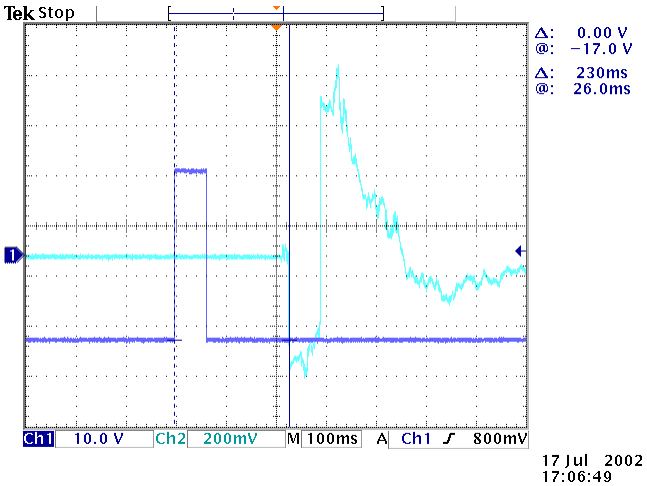
Run 10:



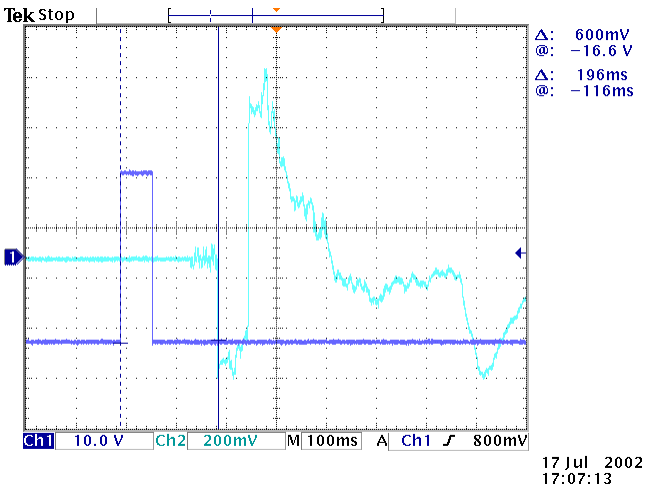
Run 11:



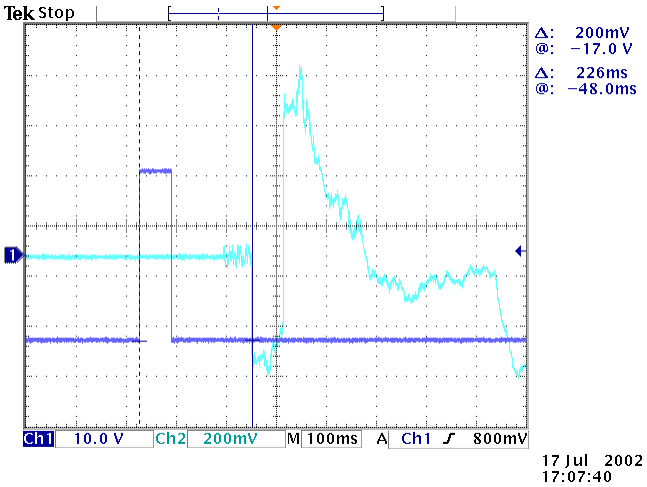
Run 12:



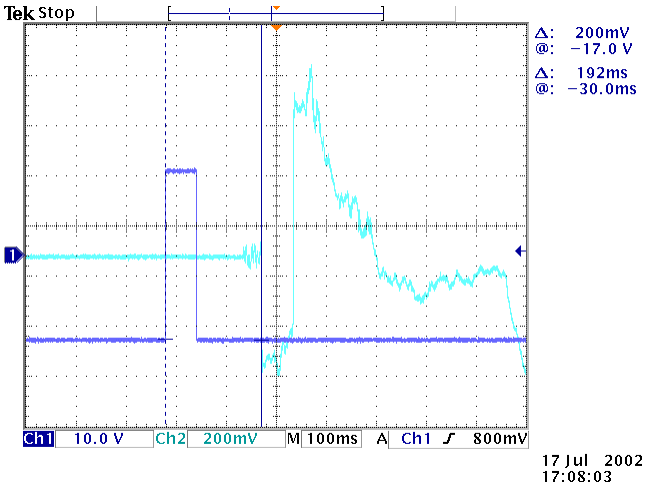
Run 13:



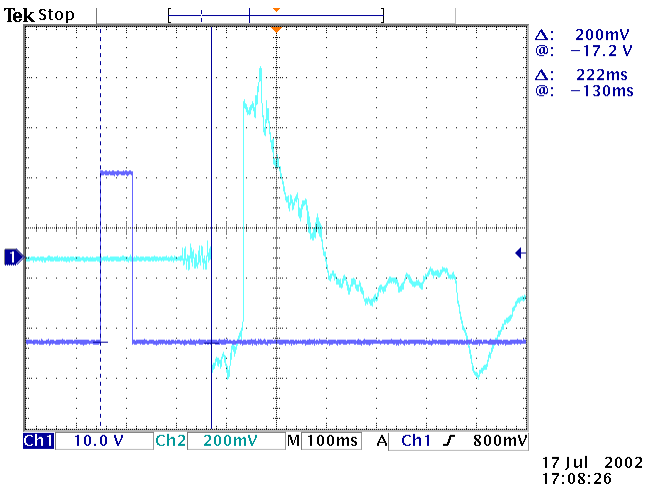
Run 14:



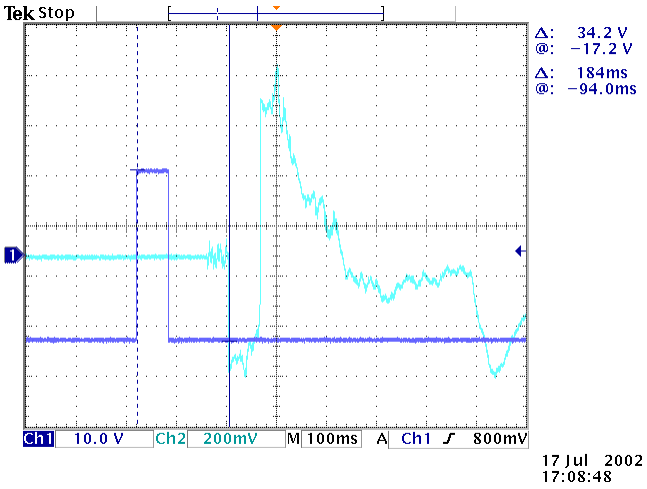
Run 15:



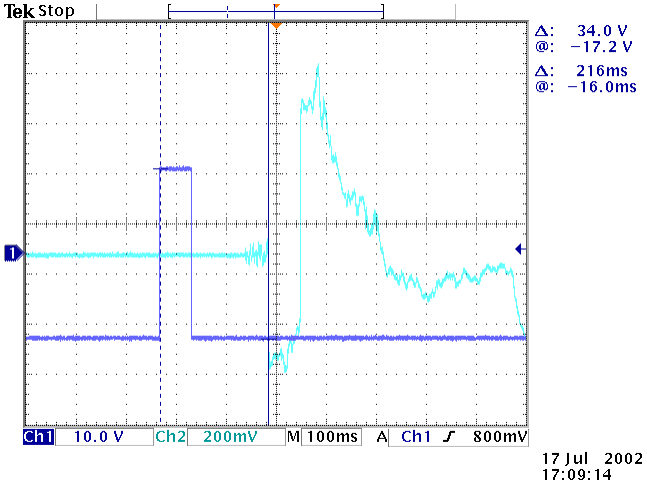
Run 16:



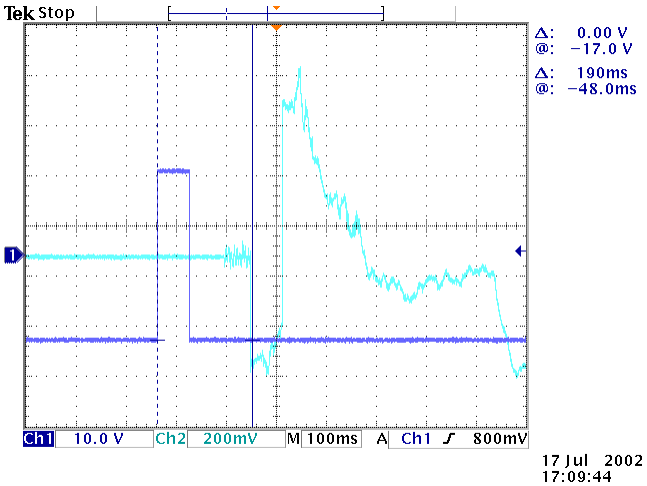
Run 17:



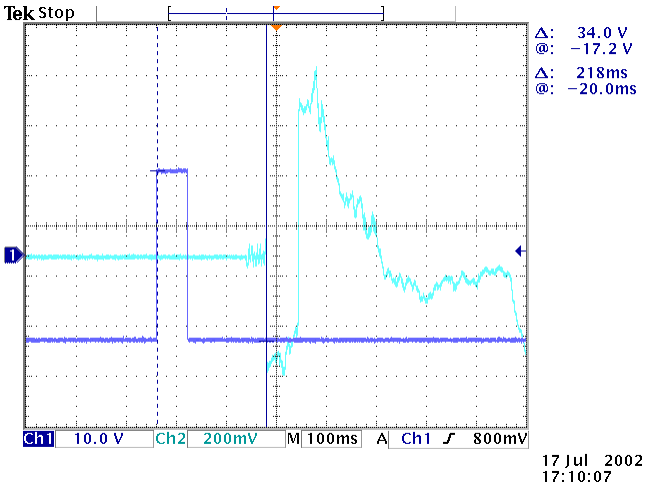
Run 18:



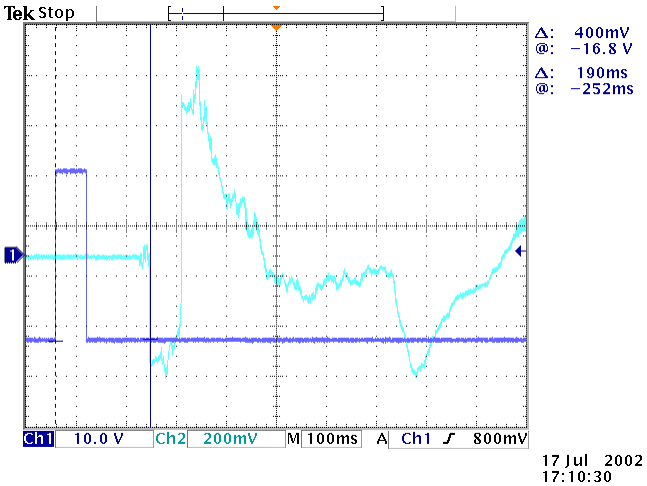
Run 19:



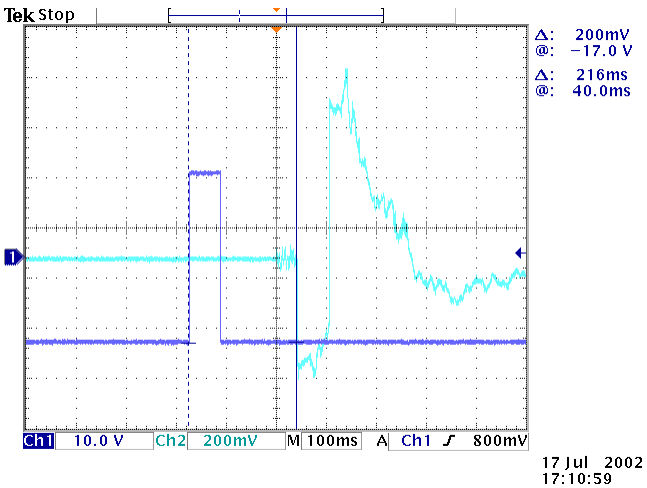
Run 20:



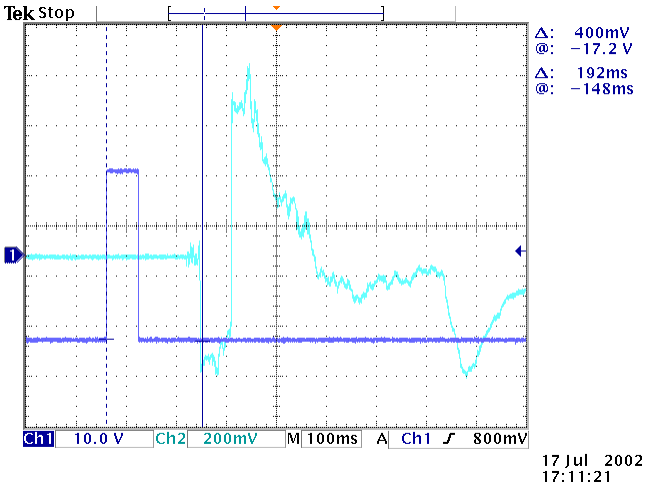
Run 21:



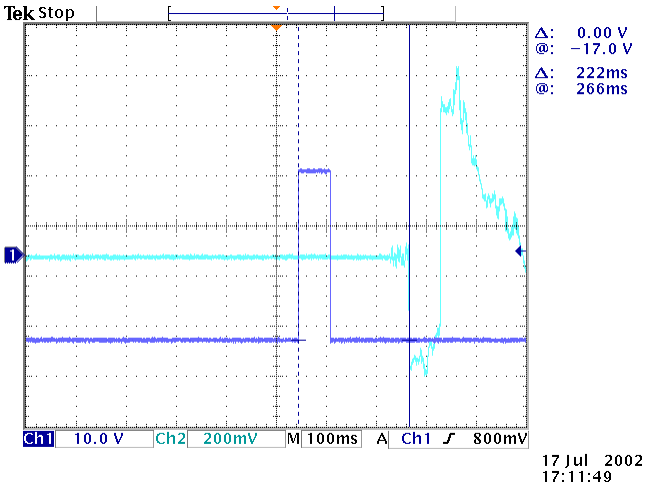
Run 22:



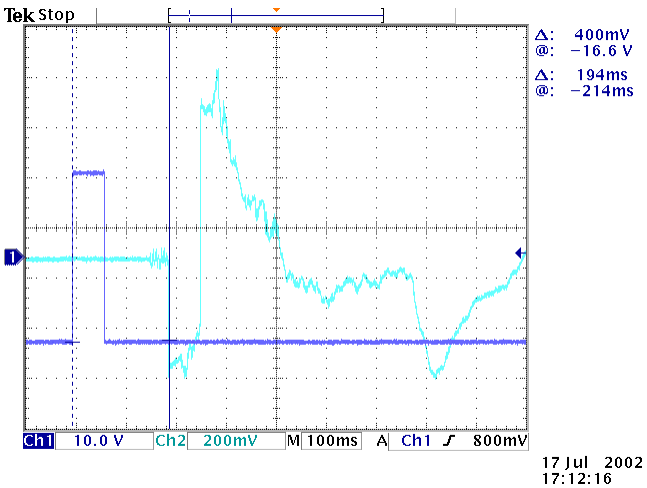
Run 23:



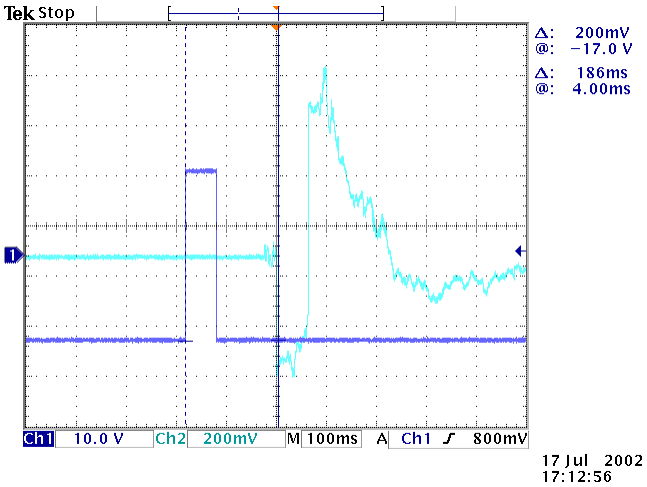
Run 24:



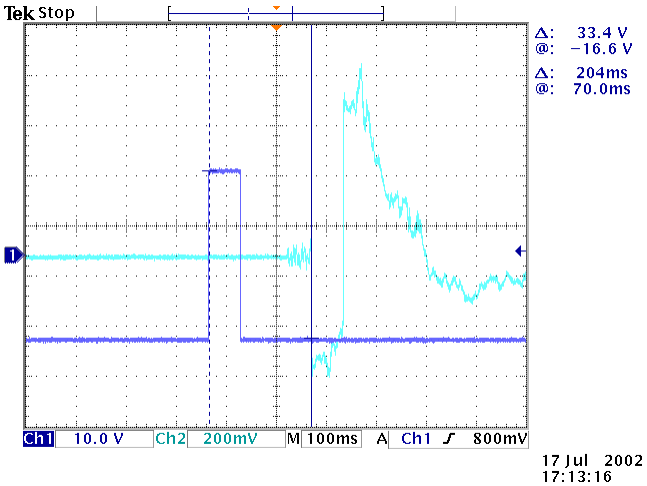
Run 25:



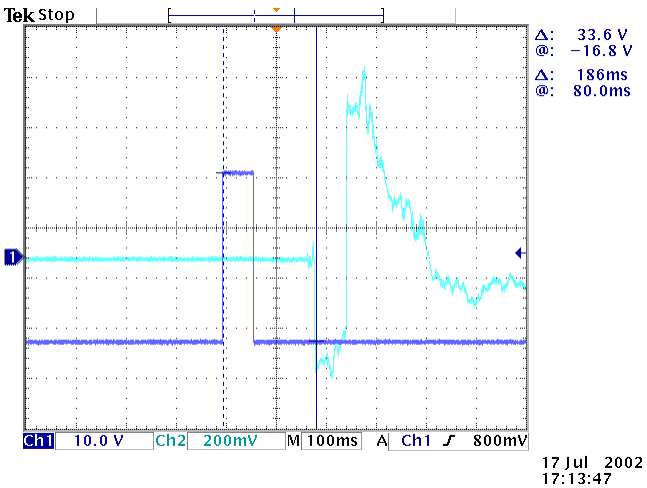
Run 26:



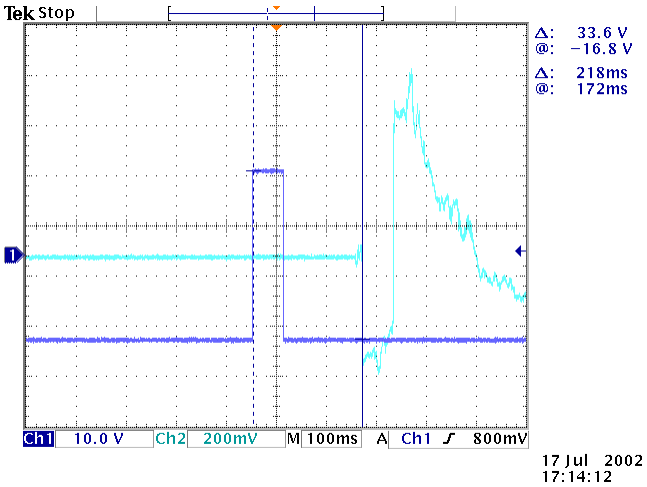
Run 27:



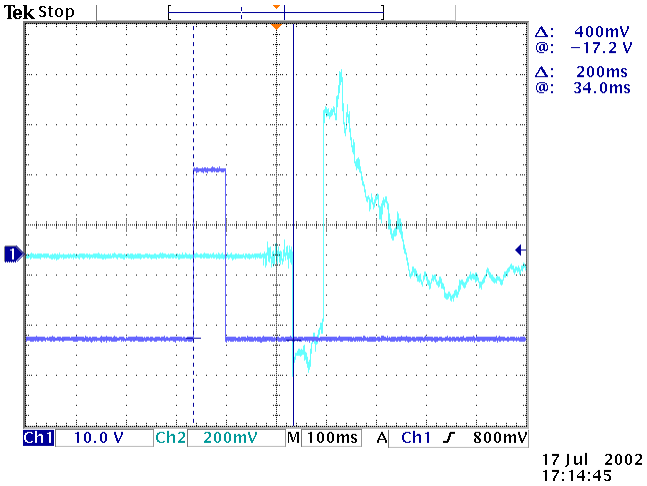
Run 28:



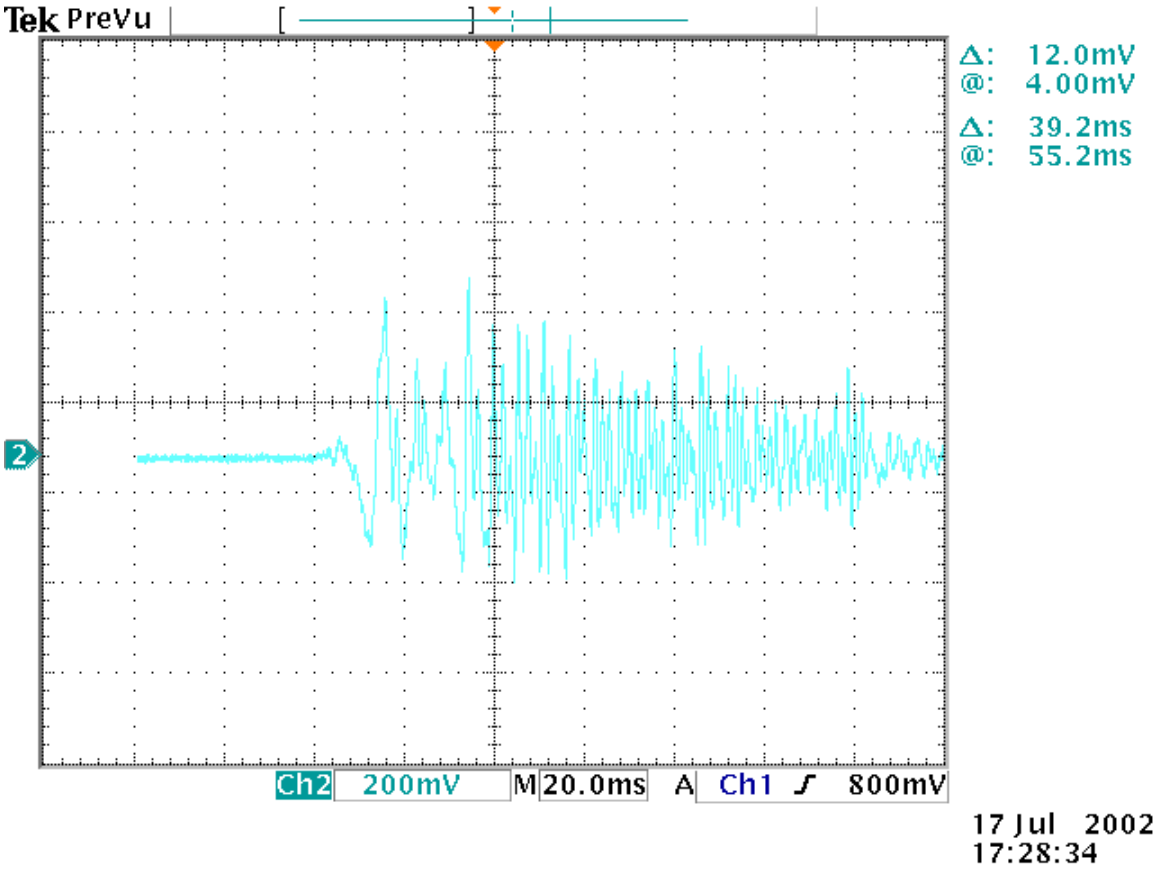
Run 29:


Tek Stop
Δ: 33.6 V
@: −16.8 V
Δ: 218ms
@: 172ms
Ch1 10.0 V Ch2 200mV M 100ms A Ch1 800mV
17 Jul 2002
17:14:12

Run 30:


Tek Stop
Δ: 400mV
@: −17.2 V
Δ: 200ms
@: 34.0ms
Ch1 10.0 V Ch2 200mV M 100ms A Ch1 800mV
17 Jul 2002
17:14:45

D. SAMPLE DIRECTVOICE LIVE VOICE



LIST OF REFERENCES

[Ausim3D, http://www.ausim3d.com]

Baldis, J. J., Effects of Spatial Audio on Memory, Comprehension, and Preference During Desktop Conferences, *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 166-173, 2001.

Begault, D. R. and Wenzel, E. M., Headphone Localization of Speech, *Human Factors*, The Human Factors and Ergonomics Society, Vol. 35, No. 2, pp. 361-376, 1993.

Begault, D. R., *3D Sound for Virtual Reality and Multimedia*, Academic Press, Boston, 1994.

Blauert, J., *Spatial Hearing: The Psychophysics of Human Sound Localization*, MIT Press, Cambridge, MA, 1974.

Bolot, J. C. and Fosse-Parisis, S., Adding Voice to Distributed Games on the Internet, *Conference on Computer Communications (IEEE Infocom)*, San Francisco, CA, 1998.

Brown, J. S., Collins, A. and Dugui, P., Situated Cognition and the Culture of Learning, *Educational Researcher*; Vol. 18, No. 1, pp. 32-42, 1989.

Brungart, D. S. and Simpson, B. D., Distance-Based Speech Segregation in Near-Field Virtual Audio Displays, *Proceedings of the 2001 International Conference on Auditory Displays*, Espoo, Finland, 2001.

Campbell, J. R., *The Effect of Sound Spatialization on Responses to Overlapping Messages*, Master's Thesis, Naval Postgraduate School, Monterey, CA, 2002.

Chi, M. T., Glaser, R. and Farr, M. J., *The Nature of Expertise*, Lawrence-Erlbaum Associates, Hillsdale, NJ, 1998.

Dinh, H. Q., Walker, N., Song, C., Kobayashi, A. and Hodges, L. F., Evaluating the Importance of Multi-Sensory Input on the Sense of Presence in Virtual Environments, *Proceedings of IEEE Virtual Reality*, Houston, TX, 1999.

EAX 3.0 Software Development Kit (SDK), [http://developer.creative.com]

Grassi, C. R., *A Task Analysis of Pier Side Ship- Handling for Virtual Environment Ship-Handling Scenario Development*, Master's Thesis, Naval Postgraduate School, Monterey, CA, 2000.

Greenwald, T. W., *An Analysis of Auditory Cues for Inclusion in a Virtual Close Quarters Combat Room Clearing Scenario*, Master's Thesis, Naval Postgraduate School, Monterey, CA, 2002.

Microsoft DirectX 8.1 Software Development Kit (SDK), [http://www.microsoft.com\windows\directx]

Nelson, W. T., Bolia, R. S., Ericson, M. A. and McKinley, R. L., Spatial Audio Displays for Speech Communications: A Comparison of Free Field and Virtual Acoustic Environments, *Proceedings of the Human Factors and Ergonomics Society 43rd Annual Meeting*, Houston, TX, pp. 1202-1205, 1999.

Norris, S. D., *A Task Analysis of Underway Replenishment for Virtual Environment Ship-Handling Scenario Development*, Master's Thesis, Naval Postgraduate School, Monterey, CA, 2000.

OpenAL Software Development Kit (SDK), [http://www.openal.org] or [http://developer.creative.com]

Posner, M., *Introduction: What Is It To Be an Expert?*, University of Oregon, 1983.

Scourgie, M. and Sanders, R., *The Effect of Sound Delivery Methods on a User's Sense of Presence in a Virtual Environment*, Master's Thesis, Naval Postgraduate School, Monterey, CA, 2002.

Wenzel, E. M., Effect of Increasing System Latency on Localization of Virtual Sounds, *Proceedings of the Audio Engineering Society 16th International Conference on Spatial Sound Reproduction*, Rovaneimi, Finland, pp. 42-50, 1999.

Wickens, C. D. and Hollands, J. G., *Engineering Psychology and Human Performance, 3rd Edition*, Prentice-Hall, Upper Saddle River, NJ, 1999.

Yewdall, D., *Practical Art of Motion Picture Sound*, Focal Press, Boston, MA, 1999.

Yost, W. A., Dye, R. H. and Sheft, S., A., Simulated "Cocktail Party" with Up to Three Sound Sources, *Perception and Psychophysics*, Vol. 58, pp. 1026-1036, 1996.

BIBLIOGRAPHY

Shilling, R. D. and Shinn-Cunningham, B. G., Virtual Auditory Displays, *Handbook of Virtual Environment Technology*, K. Stanney (ed), Lawrence Erlbaum, Associates, Inc., New York, 2002.

Shinn-Cunningham, B. G., Applications of Virtual Auditory Displays, *Proceedings of the 20th International Conference of the IEEE*, Hong Kong, China, Vol. 20, No. 3, pp. 1105-1108, 1998.

Shinn-Cunningham, B. G., Spatial Auditory Displays, *International Encyclopedia of Ergonomics and Human Factors*, W. Karwowski (ed), London: Taylor and Francis Ltd., 2001.

Wenzel, E. M., Wightman, F. L. and Kistler, D. J., Localization with Non-Individualized Virtual Acoustic Display Cues, *Conference on Human Factors and Computing Systems*, New Orleans, LA, pp. 351-359, 1991.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Dr. Michael Zyda, Chairman
   Modeling, Virtual Environments and Simulation (MOVES) Institute
   Monterey, California

4. Dr. Russell Shilling
   Modeling, Virtual Environments and Simulation (MOVES) Institute
   Monterey, California

5. Dr. Rudolph Darken
   Modeling, Virtual Environments and Simulation (MOVES) Institute
   Monterey, California

6. CDR Eric Krebs
   Fairfax, Virginia